UNIVERSITY OF OSLO
Department of Informatics

GPU Virtualization

Master thesis

Kristoffer Robin
Stokke

May 1, 2012

Acknowledgements

I would like to thank my task provider, Johan Simon Seland, for the opportunity to get dirty with bits, bytes, low level programming and virtual hardware.

I would also like to thank him, Stein Gjessing, Tore Østensen, Vera Hermine Goebel, Rune V. Sjøen and Anders Asperheim for their help in reviewing my work and making suggestions along the way.

During the course of the thesis I have been involved in Qemu's developer community. My sincerest thanks go to stefanha and pbrook on Qemu's IRC channel for pointing me in the way of Virtio, Qemu's object model and Qemu's DMA facilities. Also a big thanks to the other developers who came with tips, help and suggestions along the way!

To my fellow students at the ND lab; thank you all for a thriving environment with discussion and feedback.

Last but not least, my thanks go to family and friends who showed me their support during the work.

Abstract

In modern computing, the Graphical Processing Unit (GPU) has proven its worth beyond that of graphics rendering. Its usage is extended into the field of general purpose computing, where applications exploit the GPU's massive parallelism to accelerate their tasks. Meanwhile, Virtual Machines (VM) continue to provide utility and security by emulating entire computer hardware platforms in software. In the context of VMs, however, there is the problem that their emulated hardware arsenal do not provide any modern, high end GPU. Thus any application running in a VM will not have access to this computing resource, even if it can be backed by physically available resources.

In this thesis we address this problem. We discuss different approaches to provide VMs with GPU acceleration, and use this to design and implement Vocale (pronounced "Vocal"). Vocale is an extension to VM technology that enables applications in a VM to accelerate their operation using a virtual GPU.

A critical look at our implementation is made to find out what makes this task difficult, and what kind of demands such systems place on the supporting software architecture. To do this we evaluate the efficiency of our virtual GPU contra a GPU running directly on physical hardware.

This thesis holds value for readers who are interested in virtualization and GPU technology. Vocale, however, also features a broad range of technologies. Anyone with interest in programming in C / C++, software libraries, kernel modules, Python scripting and automatic code generation will have a chance of finding something interesting here.

Contents

List of Figures

List of Tables

Code Examples

Chapter 1

Introduction

Virtualization is an abstract concept with roots deep into IBM's mainframe history. In computing, virtualization technology attempts to create a virtual, as in "fake", version of a system, that reacts and behaves like the real one[34]. These systems can provide many benefits in form of security, utility and compatibility.

A typical, modern example is how some server companies rent out machines to their customers. Since computer hardware (and especially servers) is expensive, it is not cost effective to buy a dedicated machine for each customer. Instead, they install a smaller set of powerful machines. These emulate several Virtual Machines (VM) in software using virtualization technology. Each VM is an entire machine platform consisting of a processor, network interface, display adapter et cetra (see Figure 1.1 on the following page). The benefit of this is the illusion of having more servers for rent than what is actually physically available, in a way that is transparent to the customer.

VMs rely on *hypervisors*[27] to work. Hypervisors are complex pieces of software responsible for emulating every piece of hardware in the VM, and by extension, the VM itself. In the ever changing world of computing technology, this creates a unique problem when computer hardware arrives at the scene: Hypervisors have to implement the new hardware for their VMs.

This is especially true for the introduction of the Graphical Processing Unit (GPU). Traditionally, GPUs are designed for off-loading compute-intensive tasks related to graphics rendering from the Central Processing Unit (CPU). Over the past fifteen years, however, the GPU has evolved into a massively parallel coprocessor[22]. This means that the GPU is now also

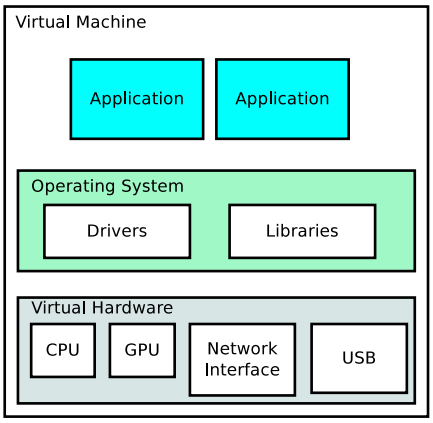

Figure 1.1: A simple virtual machine running an operating system.

used for general purpose computing in addition to its graphics related tasks. We call this General Purpose computing on GPUs (GPGPU).

GPGPU computing facilities are provided through programming architectures. An example is the Compute Unified Device Architecture (CUDA)[9] provided by NVIDIA, a GPU manufacturer. Applications and OSs are expected to use this extra computing resource to a higher extent in the coming years[26].

In the context of VMs, the problem is that no hypervisors provide them with a virtual GPU resource that can be used for general purpose computing. This would be a great thing to have, as OSs and applications running in the VM could benefit from an extra coprocessor. Server companies could rent out massively parallel computing power through virtual GPU abstractions. There is, however, little research in these fields.

The most important contribution of this work is Vocale (pronounced "Vocal"), a complex software project written as a part of this thesis. Vocale is an acronym for Virtual CUDA Library Extension. It is a collection of program extensions, libraries and drivers that attempts to bring a virtual GPU resource to VMs. It is based on a hypervisor named Qemu. As the name suggests, it provides GPGPU facilities to Qemu's VMs by emulating the CUDA software library. Vocale makes it easy to see why virtual GPU acceleration is hard to achieve, and what demands it places on software that attempts to do it.

Target readers include anyone with an interest in virtualization or GPU

technology. Others may be interested in the technical areas and challenges of Vocale, including application / OS programming, the CUDA framework, the Qemu hypervisor, Python scripting and automatic code generation.

1.1 Problem Statement

Today's hypervisors incorporate no GPU acceleration for their VMs. This presents an issue in cases where the physically available GPU resources in a machine cannot be utilized by the applications running in a VM. This thesis wants to:

- Look at previous work in the field of GPU virtualization.
- Look at possible design paradigms for and implement a virtual GPU suitable for general purpose computation in a hypervisor.
  - Identify the challenges and trade-offs of implementing this into existing virtualization technology.
  - If possible, devise new methods or proposals to overcome the challenges.
  - Study the performance impact of virtualized GPU acceleration against its non-virtualized counterpart.

1.2 Overview

The disposition of the thesis is as follows. This chapter describes the motivation and problem statement for this work.

The following chapter presents background information about virtualization technology, GPUs, kernel modules, software libraries and virtual memory. We also present a short terminology for terms used throughout the thesis. This is important to understand Vocale and the results and conclusions presented later. Finally, we have a look at previous work in the field of GPU virtualization, as well as the research and development that occurred during the work of the thesis.

The third and fourth chapters outline the design and implementation of Vocale. Vocale is an extension to the Qemu hypervisor that provides VMs with GPU acceleration, and was developed as a part of this thesis. We

discuss the design and general architecture of Vocale in the design chapter, and go into implementation specific details in the implementation chapter. This shows how Vocale accomplishes its tasks. Note that Vocale has been built in two iterations, 1.x and 2.x.

Chapter five discusses benchmark and test results of Vocale as well as the implementation itself. We argument for and describe the performance tests Vocale has been subjected to, and compare the performance of Vocale with its non-virtualized counterpart.

The final chapter presents the conclusions that have been derived from the work, and discusses opportunities for further work in this area. We state the main problems involved in the design and implementation of virtual GPUs and the demands it places on the hypervisor.

Appendix A and B contains, respectively, hardware details about our test bed and a small performance test of `virtio-serial`, a virtual hardware device used by Vocale. We will reference these when necessary.

1.3 Vocale Software

Vocale's source code can be downloaded from SourceForge at the following link location:

https://sourceforge.net/projects/vcuda/

At the back of the thesis you will also find a DVD with the software project. Refer to the `readme.pdf` file in the root folder for instructions on installation and usage. The file also contains detailed instructions on how code can be added to Qemu, and other information that has no place in this thesis (but is useful for developers).

Chapter 2

Background

This chapter gives background information about important concepts and technologies necessary to understand Vocale and the work presented in this thesis.

The chapter layout is as follows. The first section yields a short but important terminology that is used throughout the thesis.

The following section gives a thorough introduction to virtualization, explaining what it is, the advantages it provides and its variations in a historical context. It then explains hypervisors and VMs in detail.

The third section gives an introduction to GPUs and NVIDIA's CUDA framework, a GPGPU programming architecture. We outline the architectural differences between GPUs and CPUs as we describe how GPGPU technology can accelerate program operation. A short introduction to CUDA's API and programming model is given to prepare the reader for the design and implementation of Vocale's virtual CUDA library.

The fourth section gives a brief look at Linux software libraries and kernel modules. Vocale makes extensive use of these to achieve its goals, and especially kernel module programming represents a large amount of the work in this thesis.

The fifth section gives a brief look at the Memory Management Unit (MMU) of CPUs. As we will see in our results chapter, finding an efficient way to transfer large amounts of data between a VM and its hypervisor is hard. Understanding the MMU is key to understand why this is and our concluding remarks in the final chapter. It is also required for any future work to understand the MMU and appreciate the complexity it imposes on Vocale.

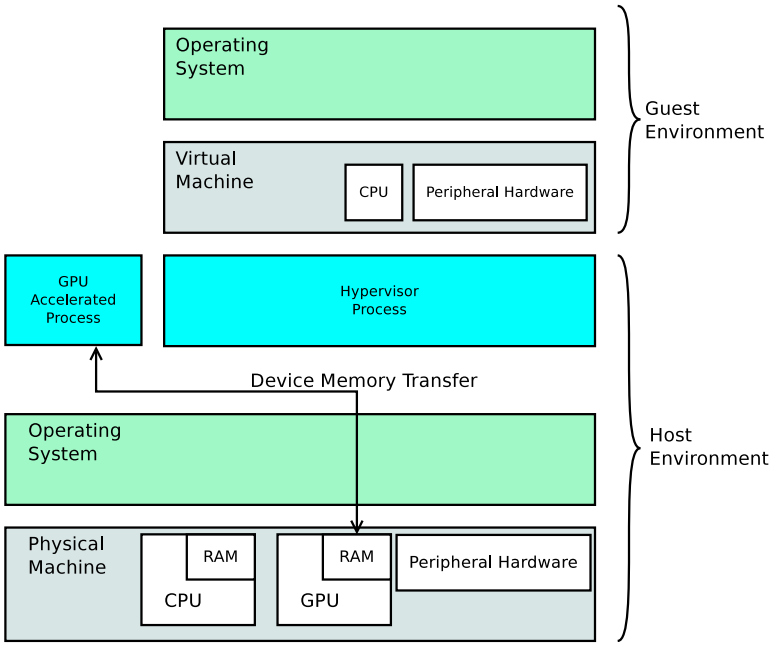

Figure 2.1: Terminology used throughout the thesis.

The final section of this chapter outlines previous work in the area of virtual GPUs and the research and development that have surfaced during the work of this thesis.

We start off by introducing important terminology used throughout the thesis.

2.1 Terminology

In this thesis, we regularly refer to the *host*, *device* and *guest*. Normally, these mean slightly different things depending on the context in which they are mentioned (GPU or hypervisor technology). Therefore it is important to define these to avoid any misunderstandings. The following list describes the most important terms used throughout the thesis (see Figure 2.1).

Host environment. The host is the physical machine in which the OS and the hypervisor are running in.

Guest environment. The guest is the VM as emulated by the hypervisor, and its OS.

Device. Denotes the physical GPU of the host.

Device memory transfers. GPUs have their own, dedicated memory. GPU accelerated applications constantly transfer data between the host and GPU memory areas. Device memory transfers encompass data transfers in all *directions*, which can be:

- Host to Device (H2D).
- Device to Host (D2H).
- Device to Device (D2D).

Note that in some cases, the host will be replaced with the guest when speaking about device memory transfers. This is natural when referring to data that is passed between the guest environment and the device.

With the terminology explained, we can start with background information. The next section introduces virtualization technology.

2.2 Virtualization Technology

This section gives a more detailed explanation of what virtualization is, before giving a brief look at it from a historical perspective. At the end of the section we look at different hypervisors, which is important for our design discussion in Chapter 3.

2.2.1 What is Virtualization?

In short, virtualization is to impose the role of another system. This process is called *emulation*. Virtualization technology creates an unreal, fake version of a system in a way that is transparent to the users of that system. Transparency is achieved by preserving the system's interface. The real benefit of this is to be able to perform the internal tasks of the system in new ways; as we will see now in the following example.

Consider an application that depends on a FAT File System (FS) to work. Such an application can be seen to the left in Figure 2.2 on the following page. Then, imagine what would happen if you had to upgrade your hard drive and

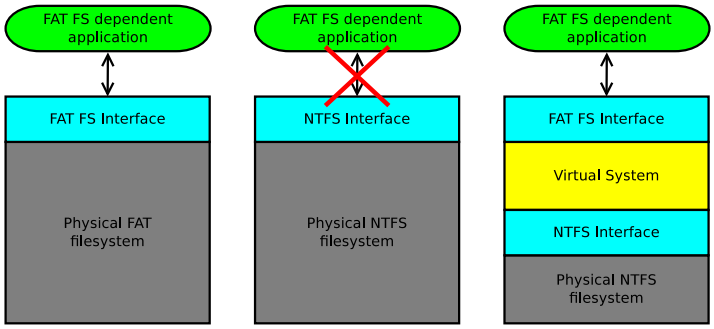

Figure 2.2: A virtualization scenario.

end up with another FS, for example NTFS. The application would now stop working. A solution to this problem could be to virtualize the FAT FS; see the right model in Figure 2.2. In the figure, the virtual system synthesizes FAT operations into NTFS operations in a way that preserves the old FAT interface. Thus the application can still work, and the trade-off is some extra overhead in FS operations.

Virtualization is not a new concept. Its first traces can be found in the old mainframe computers of IBM, used for providing backwards compatibility[19], virtual memory[18] and Virtual Machine Monitors[13] (covered in Section 2.2.3). These systems were among the first to provide hardware assisted programming and multiprogramming, which is taken for granted by today's programmers.

Virtualization is still an active technology today because of the advantages it provides; utility, compatibility and security. Java applications run on a VM, achieving compatibility as the VM is compatible with different computer architectures. The server company example from the introduction is both cost-effective and secure. Developers use VMs to test applications designed for other CPU architectures than the one they are physically using, achieving utility. Virtual memory creates the illusion of having more main memory (RAM) than what is physically available. Harmful programs can be run in a virtual environment without endangering the entire system, a technique called sand-boxing. Finally, virtualization can be used to create abstractions that allow for sharing of hardware resources, which is important in cloud computing[31].

However, most of these virtualization examples are out of scope for this

document, which focuses on hypervisors.

2.2.2 Types of Virtualization

The field of virtualization brings about some general concepts that describes the properties of a virtual system.

Full Virtualization. In full virtualization systems, the user of the system has absolutely no idea that it is running in an emulated environment. It acts and interacts with the environment in a way that is identical to running in a real system. These systems, and especially hypervisors, traditionally suffer from poor performance.

Paravirtualization denotes virtualization scenarios where the user knows that it is running in an emulated environment, as opposed to full virtualization. This can have a positive impact on performance of the system in cases where full virtualization imposes a large emulation overhead.

Hardware Assisted Virtualization refers to the usage of hardware to accelerate or help the operation of virtual systems. For example, Popek and Goldberg[23] state that for a processor architecture to be virtualizable, all privileged machine instructions must cause *traps* on failure, which is a form of exception handler. The x86 processor architecture is hard to virtualize because some privileged instructions fail without causing traps. The problem of this is that the hypervisor is never notified that its guest attempted to execute the instruction, and hence is unable to emulate it in software. To address this, Intel has developed hardware support called Intel-VT[28] to address this and four other issues. While investigation of these technologies is out of scope for this thesis, they significantly accelerate the operation of VMs.

Hardware and Software Virtualization describes the type of the system being emulated. Qemu emulates a VM (hardware virtualization), while a program called Wine emulates a Windows compatibility layer on top of Linux (software virtualization). A virtual library like the one implemented by Wine is advantageous over the alternative when considering performance, which is to install Windows in a VM.

2.2.3 Hypervisors

Hypervisors are highly complex programs that emulate entire computer hardware platforms, called VMs, in software. The VM incorporates an emulated CPU, hard drive, physical memory, network interface, display adapter (GPU) et cetra. The VM can be used just like a physical machine.

The grandfather of modern hypervisors can be found in the CP/CMS research project[13] from the mid 1960s. It is interesting to have a quick look at this system not only because it relied on virtual systems such as virtual memory and virtual disks, but also provided a virtual system in itself. It consisted of an S/360 model 40 mainframe (see Figure 2.3 on the following page) featuring a Control Program (CP) and a Conversational Monitor System (CMS).

The CP was called a Virtual Machine Monitor, but is most similar to today's hypervisors in that it could emulate VMs in software. Combined with CMS, a form of single user interactive OS, it gave users simultaneous access to a machine, and was one of the first systems to be able to do this. This old project provided increased utility and security by letting each user work in an isolated environment, unable to affect and harm others.

Hypervisor Design Strategies

Hypervisors are designed according to two common design paradigms; Type 1 and Type 2[27]. Type 1 hypervisors emulate the machine from inside an OS. This approach is normal for users who want to stay inside their everyday OS and machine architecture, but also want the flexibility of being able to run or test programs designed for another. Figure 2.4 on page 12 shows such a system, where an OS (the guest) is running on top of another OS (the host) through a virtual machine. It is normally possible to emulate multiple virtual machines in parallel. Type 2 hypervisors run directly on the hardware in place of the OS, see Figure 2.5 on page 12. This typically puts a type 2 hypervisor at disadvantage since it has to implement its own hardware drivers. A type 1 hypervisor can benefit from the hardware management given by the hosting OS[20].

When emulating a VM, the hypervisor has to impose a computer hardware platform from the time it is booted to the time it is shut down. Most hypervisors offer some hardware configuration options, like the amount of RAM to be emulated, the processor architecture, disk size et cetra. The

Figure 2.3: The IBM S360 model 40 mainframe. Courtesy of the IBM archives.

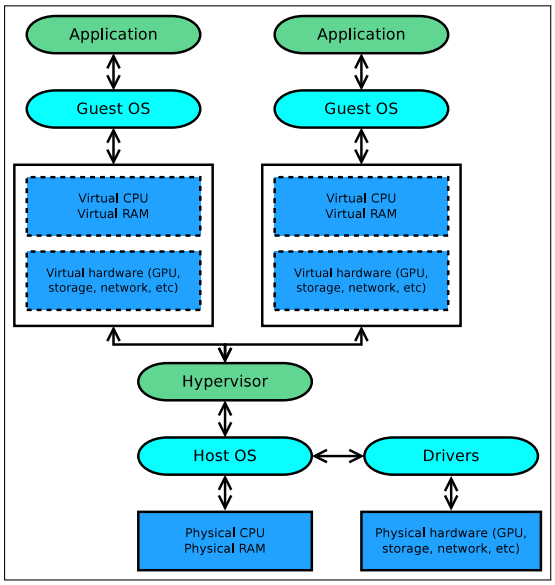

Figure 2.4: A type 1 hypervisor.

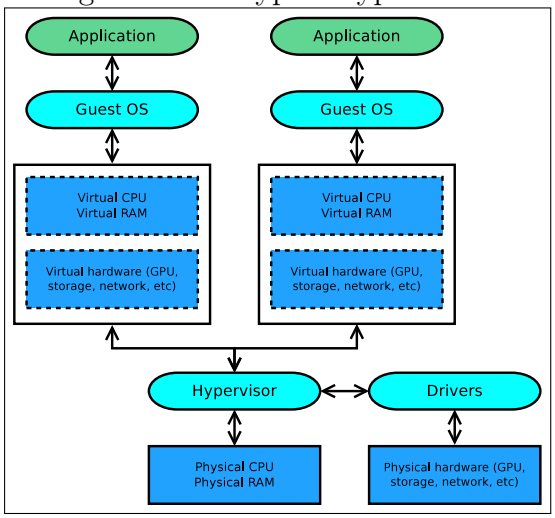

Figure 2.5: A type 2 hypervisor.

hypervisor has to emulate calls to the peripheral devices and the target processor architecture according to the correct behaviour.

Following are some common ways to perform this emulation. The virtual device mentioned can be anything; a CPU, virtual disk et cetra.

1. Calls to the virtual device are passed modified or directly to a corresponding physical device and run natively.

2. Calls to the virtual device are emulated using hardware assisted virtualization support, for example Intel-VT[28] or AMD CPU's equivalent, AMD-V (see Section 2.2.2).

3. Calls to the virtual device are emulated in software.

For example, consider a hypervisor emulating a virtual disk. Its main responsibility is to emulate the interfaces of the disk, for example by implementing a disk driver in the VM. Internally, the actual writing to the disk is done to a file within the host's file system. When a guest OS is writing to its virtual disk, it is happily unaware that it's actually writing to a file - the guest believes that it is in complete control of its own hard drive.

Hypervisor Software

The following list describes some common open source and proprietary hypervisors. This is important for the design discussion in Chapter 3.

Xen is a type 2 hypervisor that uses the paravirtualization concept. It runs directly on the physical hardware of the machine, while the guest OSs run in parallel on top of the hypervisor. Note that it uses a special OS, designated `dom0`, to mediate access to the hardware. Guest OSs must be modified to be able to run efficiently. This is Xen's way of coping with the uncooperative x86 architecture (refer to Section 2.2.2 on page 9). It uses Qemu, described below, to emulate the hardware platform[3, 20].

VMware hosts a number of proprietary virtualization products but also an open source hypervisor called VMware player. This type 1 hypervisor runs on top of the host OS, and is able to simulate several guest OS environments in parallel. It runs on Linux and Windows, and supports unmodified guest OSs.[35].

VirtualBox is also a type 1 hypervisor. It supports hardware virtualization and uses Qemu to emulate the hardware platform. One of the advantages of VirtualBox is its independency of hardware virtualization support, making it a viable alternative for older computer hardware as well as new. It is also well documented and supports Linux as the host and guest OS[11].

Qemu is a type 1 hypervisor using the Linux kernel to accelerate performance. It makes use of hardware assisted virtualization and a special kernel module called `KVM` (Kernel-based Virtual Machines). This gives Qemu impressive performance. Qemu is also used by other hypervisors, such as Xen, to emulate a hardware platform. [32, 33, 20].

With our introduction of virtualization technology, we can now proceed with the next section which describes GPU technology.

2.3 Graphical Processing Units

This section introduces the GPU. We outline the parallel nature of the GPU in a historical view, and show how the GPU has become a general purpose coprocessor. The following subsection explains CUDA, where we present CUDA's programming model, its main architectural concepts and its APIs. It is important to understand how developers interact with CUDA to realise what Vocale needs to do to emulate it.

Modern GPUs are massively parallel devices dedicated to generating and drawing graphical objects on a screen. It sees its daily use as a coprocessor to the CPU by offloading graphics related processing. GPUs are necessary in scenarios where the CPU cannot be expected to satisfy all the computational demands related to graphical rendering, networking, program flow, interrupts et cetra. Examples include gaming scenarios and live video encoding / decoding.

The need for a dedicated graphics device is marked by the introduction of a certain piece of software called Sketchpad. Sketchpad was a computer program that introduced many of the modern Graphical User Interface (GUI) concepts used today. It was developed by a PhD student at MIT named Ivan Sutherland as part of his PhD research in 1963, and provided an interface to draw, copy, move and describe objects on a screen. It is considered the dawn of graphical computing and had huge effect on this research area in

later years. In 1968, Ivan Sutherland and David Evans founded Evans & Sutherland, a pioneer company creating graphics hardware for the computer systems at MIT.

Up until then, graphics devices were purely vector based in that they accessed vector endpoints and drew the corresponding lines on a monitor. This changed with the introduction of raster graphics, where each pixel in the display device is represented by an amount of bits in memory. Jim Clark, who had worked at Evans & Sutherland, co-founded another graphics hardware company in 1981 called Silicon Graphics (SGI). SGI is famous for Jim Clark's Geometry Engine, which did many of the operations required to display an image on a screen. This was an important development in computer graphics hardware, as the graphics requirements of software was steadily increasing, and the main processor could not be expected to do all the graphics-related work by itself. The Geometry Engine solved this by off-loading the graphics related tasks from the main processor[4].

GPUs incorporate a *graphics pipeline*, which describe the steps taken when generating a viewable image from some kind of input from the CPU. Typically, it consists of the following steps[22].

- Triangle generation. Vertices from the objects that are to be drawn on screen are generated as triangles of vertices.
- Rasterization. This is the operation of mapping triangles into pixel locations. A pixel generated from a triangle is called a fragment.
- Fragment operations. Pixel colors are obtained from the fragments covering the pixel and other color sources such as textures.

This pipeline has become broader and more flexible as GPU technology evolved by offloading more work from the CPU and allowing certain stages, such as texture mapping and pixel shading, to be programmable. The tasks involved are also very suitable for parallel execution; which is the reason for the parallel nature of the GPU.

In later years, researchers found ways to exploit the programmability of the graphics pipeline and utilize it for general purpose computing. This has led major GPU manufacturers like NVIDIA and ATI to provide programming frameworks for their GPUs, effectively levering the GPU into a massively parallel general purpose coprocessor.

Examples of such programming framework are NVIDIA's CUDA framework, and the Khronous Group's Open Computing Language. In the following subsection, we describe CUDA's programming model and API. This is to prepare the reader for the design and implementation of Vocale.

2.3.1 CUDA - a GPGPU Framework

CUDA is a general purpose programming architecture for NVIDIA's GPUs. When applied correctly, it can be used to offload tasks from a CPU to decrease the runtime of a task, which is often the objective of optimization.

To understand CUDA it is important to understand the difference between CPUs and NVIDIA's GPUs. From a hardware perspective, the heart of the GPU consists of special purpose microprocessors call Streaming Multiprocessors (SM). GPUs incorporate a varying number of SMs; it can be anything from 10 to 1500 SMs depending on the product.

SMs have a much smaller instruction set than CPUs, but can run up to eight threads in parallel as long as they follow the same execution path. That means more than 10000 threads running simultaneously for high end GPUs. Each thread runs the same program but with different data. This architecture is called Single Instruction, Multiple Thread (SIMT) as opposed to Single Instruction, Multiple Data (SIMD) found in the MMX/SSE facilities of normal Intel CPUs. SMs are also called CUDA Cores.

Table 2.1 on the following page shows the main differences between GPUs and CPUs.

The GPU does not have any protection mechanism such as virtual memory or instruction privilege rings. It is also worth noting that GPU processing follows strict performance rules in order to achieve all the benefits of the GPU listed in Table 2.1 on the next page [10].

- All eight threads running on an SM must be *non-divergent*, that is, follow the same program execution path.
- Memory throughput is high only when *memory coalescing* is achieved, that is, special memory access patterns must be followed.
- Instruction throughput is high only when certain register *register dependencies* are met.

	GPU	CPU
Cores	Massively parallel, using many SMs, each of which can run many threads.	Parallel, but to a much smaller degree. Uses a small number of cores.
Instruction Throughput	Very high instruction throughput due to the number of simultaneously running threads.	Lower instruction throughput.
Instruction Latency	Higher instruction latency; the SMs are clocked at a lower frequency.	Low instruction latency due to the high clock frequency of each core.
Memory Throughput	High memory throughput (that is, transfer to and from the SM registers).	Lower memory throughput.
Specialization	Specialized for problems that can be solved in parallel as blocks of small, identical threads.	General purpose, supporting a more versatile range of non-identical programs that are solved sequentially.

Table 2.1: The main differences between GPUs and CPUs.

CUDA provides the means to program this device architecture in languages such as Fortran, C and C++ in a way that is general to the range of NVIDIA GPUs. This is beneficial in that no specialized knowledge of the individual GPU is needed.

When you develop with CUDA you attempt to find parallelism in your problem. For example, adding two vectors can be considered parallel in that you can simultaneously add the corresponding vector elements (see Figure 2.6).

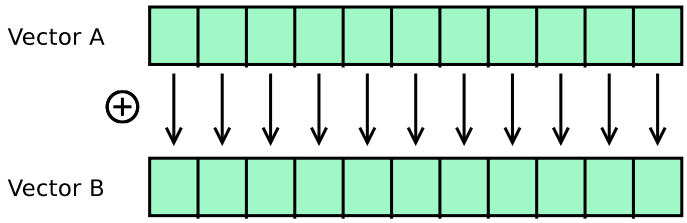

Figure 2.6: Vector addition as a parallel problem.

The next step is to adapt your parallel problem to the GPU. A CUDA developer will typically write a small GPU program called a *kernel* that performs a single element addition. The kernel is launched as a group of threads, each thread performing a single element addition on different data. A typical program flow is as follows:

1. Transfer the vector data to the device.
2. Execute the kernel as a group of threads. Each thread performs a single element addition; the number of threads depend on the size of the vector.
3. Transfer the results back to the CPU's memory.

It is important to note that we must transfer data between the device and the host's memory. We will see an example kernel to perform vector addition (code example 2.4 on page 23) at the end of this subsection.

We will now explain the fundamental programming concepts of CUDA, in particular kernel launches and the API categories. This is necessary to understand how Vocale provides a virtual CUDA library. We assume that the reader is familiar with compilation and linking of executables.

Compilation and Linking

CUDA source files have the `.cu` suffix and consists of a mix of host and device code. Device code comprise code that is sent to the GPU for execution, and host code encompass code that executes normally on the CPU. They are distinguished by prefixing symbol names with `__device__` or `__global__` for device code, and `__host__` for host code.

Device code is executed on the GPU as Parallel Thread Execution (PTX) code or architecture specific binary code[8, Appendix G]. PTX code is a unified assembly instruction set for all CUDA devices. It is guaranteed to be supported by any current or future CUDA device, regardless of the GPU's architecture. When PTX compiled device code is launched, the CUDA driver will perform just-in-time compilation to architecture specific code. Developers can also steer compilation of device code to architecture specific (binary) code to avoid this.

Normal compilers such as `gcc`, the GNU compiler collection, does not know what to do with the declaration specifiers for host and device code, nor does it support compiling PTX or architecture specific code. To cope with this, CUDA provides its own compiler, `nvcc`[7], that handles compilation of CUDA specific code. The rest is left to the host compiler, which is normally `gcc`. Linking is performed without the need of external tools.

The CUDA API

The CUDA API is split in two: The driver API and the runtime API. Both provide similar functionality, but the runtime API is more easy to use, while the driver API is harder (but provides more fine grained control). Moreover, the runtime API is built on top of the driver API. This means that some driver API functions are called implicitly when using the runtime API, for example initialization and context routines.

The APIs contain functions for managing different aspects of the CUDA library and any GPUs resident on the system. The following list describes the major API categories, but readers are referenced to the CUDA programming guide[8] and API reference manual[6] for full details. Some categories are also omitted as they have not been tested in Vocale.

The CUDA runtime API is referenced through the `cuda_runtime.h` header file, and contains the following main API categories.

- **Device Management.** Get device properties, set the current device,

reset devices and others.

- **Error Handling.** Almost all API functions return an error type which is specific to the API it belongs to. The exception is for example kernel launches, and it is necessary to use functions like `getLastError(..)` from this category.

- **Stream Management.** Many functions in the CUDA API, like kernel launches and device memory transfers, can be run asynchronously in a stream. Functions run in streams return immediately and are executed sequentially in their own thread. This category provides functions to create, query, destroy and synchronize streams.

- **Event Management.** Events can be used to record certain happenings in time. For example, an event can record when all or some of the operations in a stream are complete to profile code.

- **Execution Management.** Functions for managing kernel launches. Apart from the Execution Configuration Syntax (ECS) covered below, kernels can be launched manually using a series of function calls in this category.

- **Memory Management.** This category is quite extensive and covers a range of functions. These mainly enables us to allocate, transfer and free device memory with 1 - , 2 - and 3 - dimensional data abstractions. Memory can also be transferred asynchronously using streams.

The CUDA driver API can be referenced through the `cuda.h` header file. It contains the following main API categories implemented and used by Vocale.

- **Context Management.** SMs provide no memory protection in hardware like the virtual memory facilities of conventional CPUs. Instead, CUDA uses a software solution to prevent CUDA processes from accidentally reading or writing each other's memories: Contexts. Contexts can be thought of as a process' virtual memory; outside it, a process' kernels and device memory pointers have no meaning.

- **Module and Execution Management.** Provides, among other things, more fine grained control to load and launch kernels. Kernels

can for example be loaded from binary files or directly from memory pointers.

Kernels

At the heart of CUDA are the kernels. A kernel is a small program that is launched on the GPU as several blocks of threads. Figure 2.7 shows a 3 - dimensional master block with all the threads executing on a GPU. Each thread represents the same kernel and has its own 3 - dimensional *blockID* and *threadID*.

blockID specifies which *thread block* this thread resides in. The black squares in Figure 2.7 correspond to a single thread block.

threadID specifies the position within the thread's thread block.

Together, these are unique to each thread and are used to do thread specific operations like memory accesses. Note that the software designer is free to decide to work in the 1 - or 2 - dimensional space as well.

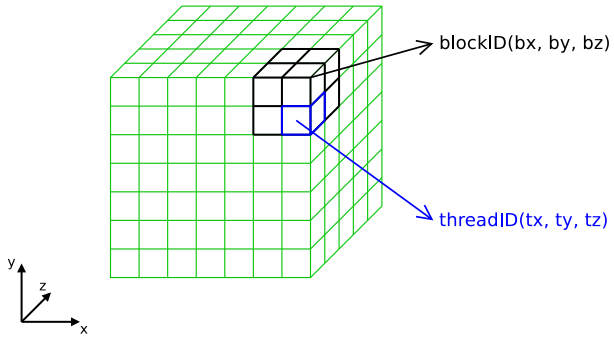

Figure 2.7: Illustrative figure of kernel threads executing on a GPU.

The programmer has to decide the number and size of the thread blocks when executing kernels, which is embedded in the kernel launch. There are three ways kernels can be launched:

- Using the ECS provided by the runtime API; see code example 2.1 on the following page.

```
yourKernelName<<<numThreadBlocks, threadsPerThreadBlock>>>
    (
        parameter1,
        parameter2,
        ...
    )
```

Code Example 2.1: Launching a kernel using the ECS.

```
cudaConfigureCall(dim3 gridDim, dim3 blockDim, size_t sharedMem,
    cudaStream_t stream)

cudaSetupArgument(const void *arg, size_t size, size_t offset)

cudaLaunch(const char *entry)
```

Code Example 2.2: Launching a kernel using the execution control functions from the CUDA runtime API.

- Directly using the execution control functions provided by the runtime API. These are relatively straightforward to use, but does the same work as the simpler ECS. They can be seen in code example 2.2.

  These are called in sequence for each launch. `cudaSetupArgument(..)` is called as many times as there are arguments to the kernel.

- Directly using the execution control functions provided by the driver API. This is the most flexible but hard to use. It supports importing device code from files and memory. Their function signatures can be seen in code example 2.3 on the following page.

As a very simple demonstration, consider code example 2.4 on the next page, which simply adds two vectors of size N using CUDA. The kernel, `vectorAdd(..)`, is launched on the GPU with some memory pointers to the vectors as arguments. The number of threads launched corresponds to the number of elements in the vector. Note the use of the th read ID to access the correct memory areas. A real kernel would involve a block ID as well, but this is omitted for the sake of simplicity.

This sums up our presentation of CUDA. This section has given an introduction to the CUDA GPGPU framework, which is important to understand Vocale's job of virtualizing the CUDA library. In the next section, we will outline the use of shared libraries and kernel modules.

```
cuLaunchKernel(
            CUfunction f,
            unsigned int
                gridDimX,  gridDimY,  gridDimZ,
            unsigned int
                blockDimX, blockDimY, blockDimZ,

            unsigned int sharedMemBytes,

            CUstream hStream,

            void **kernParams, void **extra
    )
```

Code Example 2.3: Launching a kernel using the execution control functions from the CUDA driver API.

```
__global__ void vectorAdd(
        int *inVector1,
        int *inVector2,
        int *outVector){

    outVector[threadIdx.x] =
        inVector1[threadIdx.x] + inVector2[threadIdx.x];
}

int main(){

    /* Initialize vectors of length N*/

    /* Transfer vectors to device memory v1, v2 and v3 */

    vectorAdd<<<1, N>>>(v1, v2, v3);
}
```

Code Example 2.4: A simplified kernel to perform vector addition.

2.4 Linux Libraries and Kernel Modules

The work presented in this thesis revolves a great deal around libraries and drivers. Vocale is built in several iterations and involves three drivers and two libraries, so it makes sense to give a short introduction to these concepts. Again, we assume that the reader is familiar with compilation and linking of executables in Linux.

2.4.1 Libraries

Libraries are collections of code, usually a compiled collection of subroutines and symbols, that can be linked with other pieces of code. There are three types of libraries in Linux[29]:

Static libraries. These libraries have the `.a` (archive) suffix, and are linked together with the executable in the linking phase.

Shared libraries. They have the `.so` (shared object) suffix. They are shared between programs, and are not linked before the referencing executable is run.

Dynamic libraries. Dynamic libraries are libraries that can be linked at any time during the run of the application.

The libraries developed in Vocale are of the shared type, so the rest of this section is devoted to them.

Using, Naming and Installing

Shared libraries follow a special naming convention. They have a *real name* and an *so-name*, in addition to a library reference name. These provide developers with the means to version, expand and override libraries.

Figure 2.8 on the following page shows the relationships between all these names in a simple example. The first line is an invocation of `gcc`, requesting the compilation and linking of the all-so-known `helloworld.c`. In addition to this, it has instructed the linker to include a shared library, `sample`, in the linking phase.

The so-name is simply the library reference minus the preceding '-l', adding 'lib' as prefix and '.so.x' as the suffix, where x is a version number. When `helloworld` is executed, a loader program is run that starts

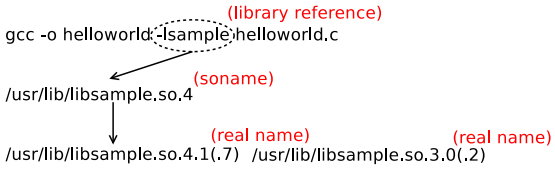

Figure 2.8: The relationship between library references, so-names and real names.

searching for the so-name with the highest version number in a predefined list of directories. This file is actually a link to the real name, which is the name of the file that contains the actual code. The loader then links this library to the executable and runs the program. The real name is denoted by the so-name plus two more versioning numbers, the last of which can be omitted.

Installing shared libraries is a simple matter. The developer just needs to

- Keep track of version numbers in the real name.
- Compile the library as shared, position independent code.
- Copy the library to one of a predefined set (`/etc/ld.so.conf`) of library folders (for example `/usr/lib`).
- Run `ldconfig`. `ldconfig` sets up the so-names and linking as necessary and keeps a cache file with links for quicker access to libraries.

Readers are referenced to [29] for more information, or to look up the makefile for Vocale's virtual CUDA library (section 4.2.2).

We have now presented shared libraries, which Vocale uses to implement the virtual CUDA library. The next section gives a brief overview of kernel modules, which Vocale also implements.

2.4.2 Kernel Modules

Kernel modules, also called device drivers, are dynamically loadable pieces of the Linux kernel. They are privileged modules of code that typically control and interact with some kind of hardware device. For example, there are kernel modules that control your network card, graphics card and hard drive. Some

kernel modules can also be controlled from user space. Under the `/dev` folder in Linux you will always find a number of such kernel modules, represented as special files in the file system (like `zero, urandom` and `null`).

Hypervisor technology will always involve kernel modules in some way since they emulate a lot of hardware. Virtual hardware is also the natural way to pass information in and out of VMs.

Kernel modules, like libraries, come in three groups[5]. They are grouped by the abstract format of the data they handle and the way in which they are accessed.

Character devices. These are accessed as streams of bytes. Examples include serial and parallel port drivers. They are represented as special files in Linux's file system.

Block devices. These are accessed by blocks of bytes, and control block oriented hardware like optical media and hard drives. Like character devices, they are represented as special files.

Network devices. Packet oriented devices. Unlike the other driver types, they are not represented by files, but instead referenced by name (for example `wlan0`).

This thesis only uses character devices, but we do not continue describing them here. Kernel module development in general is a very complex process. It involves low level interfaces to hardware and the Linux kernel, and each module type has different programming architecture. For now, it suffices to know what they are. Interested readers are highly encouraged to read [5], as well as look up the drivers developed as parts of Vocale. We will go in a bit more detail about our driver implementation in Section 4.2.1.

With kernel modules explained, our next section revolves around the MMU. Understanding this component is important to understand the amount of copying and difficulties when exchanging data between hosts and guests.

2.5 The Memory Management Unit

The work done in this thesis requires the introduction of the MMU. This electronic device makes a lot of trouble for VMs, especially in the context of

exchanging data between a VM and its hypervisor. Understanding the MMU is also important for anyone who wants to work further on Vocale.

The MMU is a part of all modern CPUs' base electronic circuitry. In a nutshell, the MMU is a hardware device that OSs use to implement virtual memory and paging. As stated in Section 2.2.1, this is an abstraction for processes and OSs that creates the illusion of running in a private, separate address space.

Figure 2.9 shows two processes running in virtual memory, also called the *user address space*. They can access the same memory pointers without bothering the other, as the OS maps the virtual addresses to different physical addresses using the MMU.

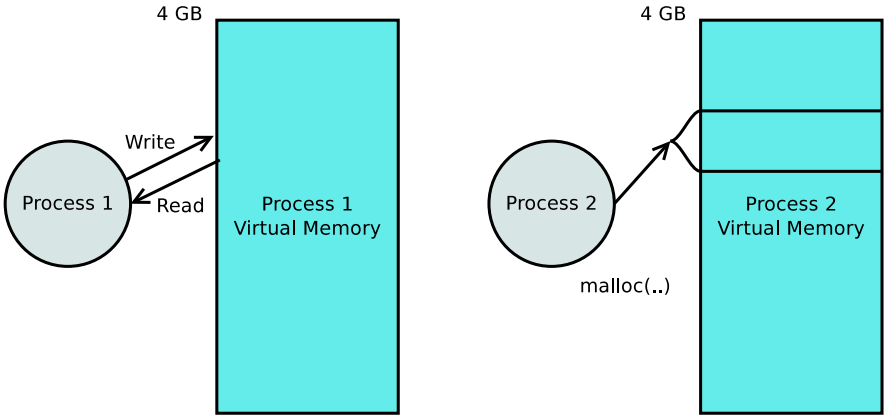

Figure 2.9: Two processes running in separate, virtual address spaces.

Having this mapping results in several benefits that are typical to virtualization technology.

- Utility. Systems can create the illusion of having more main memory than what is physically available. The MMU makes it possible to map virtual memory to external storage devices.
- Security. Processes run in their own environment with separate address spaces and are protected from each other. A process' memory buffers have no meaning in another, as it can only be accessed in the address space to which it belongs.

The *mapping mechanism* provided by the MMU is called paging. The paging mechanism is controlled by the OS and involves some rather complex work with data structures and CPU registers. In the next subsection, we give a brief overview of this system as it stands on x86 architecture. It is good to know how these things work to understand our remarks on *zero copy* guest / host data exchanges presented in the final chapter. It is not required to read this to understand most of the thesis; so readers may just want to read the bullet points on the end of the subsection.

2.5.1 Paging

From the MMU's point of view, physical memory is split into pages of a fixed length. The length can vary depending on the OS implementation, but the common size is 4096 bytes.

The challenge of the MMU is to map virtual memory addresses to physical ones on a per process basis. To achieve this, each process is equipped with:

- Many page tables referencing physical memory. A process can potentially keep page tables for its entire 4 GB span of virtual memory.
- A single page directory referencing all the page tables.

Both data structures are 4096 bytes in length, exactly the size of a physical page. Figure 2.10 on the following page illustrates the translation from virtual addresses to physical ones. The mapping is done as follows (we use VA for virtual address and PA for physical address):

1. The process keeps a pointer to the base address of its page directory. This information is handled by the OS and is kept invisible to the process.
2. When the process makes a read / write access in its virtual address space, the MMU looks up the page directory for the process.
3. The first ten bytes of the accessed VA corresponds to a 4 byte offset within the page directory. The MMU looks up this location and finds (among other, technical things) a base address pointer to one of many page tables.

4. The next ten bytes of the VA corresponds to a 4 byte offset within the page table referenced by the page directory entry in the previous step. The MMU looks up this entry and finds an offset to a physical page.

5. Finally, the remaining 12 bytes of the VA are used to offset the physical page address into a PA.

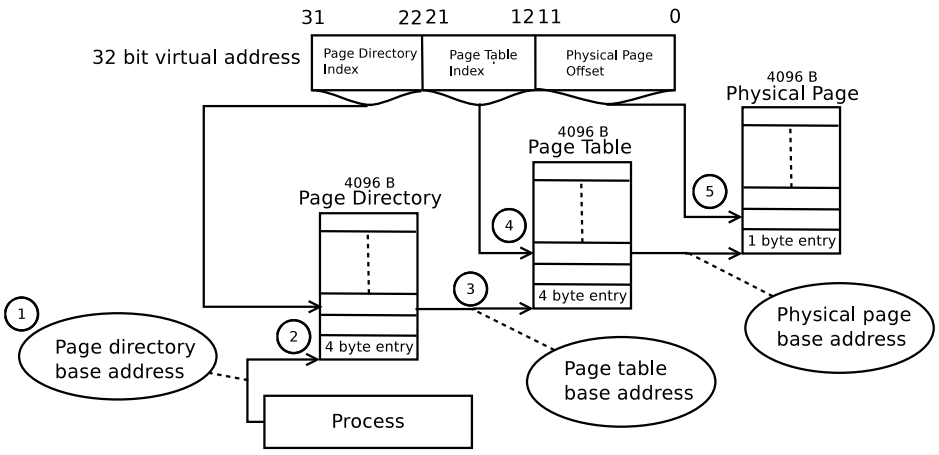

Figure 2.10: Physical to virtual address mapping using the MMU.

We have shown above how VAs are mapped to PAs by the MMU, but we have not mentioned swapping. To make the picture even more complex, physical pages can and will get swapped out to an external storage media at arbitrary times. If a page is not in the physical memory of the CPU when accessed, it will have to get swapped in.

To make it easier for the reader, we sum up some key points that are important to remember:

- Hardware devices do not have access to the MMU used by processes. As a result, if hardware wants to grab for example virtual memory of applications, the OS must be involved to map virtual addresses to physical ones. These mappings are called *scatter / gather* mappings[5, Chapter 15, page 450].

- A given physical page is not guaranteed to reside in main memory. This is an issue when it comes down to Direct Memory Access (DMA). If a

hardware device wants to access virtual memory, the OS must somehow be involved to lock memory in place.

- Even though a process "sees" a contiguous, virtual memory layout, the memory is always scattered around in physical memory. This is called memory fragmentation. Hence, hardware devices do not usually have the luxury of accessing large, continuous chunks of data.

This concludes the background information presented in this thesis. The next and final section outlines previous work in our research field.

2.6 Previous Work

This section outlines previous work in the field of GPU virtualization. While there was relatively little work in this area at the time of the start of the thesis, some approaches were already proposed, and more have surfaced as work progressed.

The first subsection gives a brief explanation about virtual GPU design paradigms given by VMware. The second subsection explains GViM, an approach to a virtual CUDA library that Vocale is similar to. The last subsection outlines some of the development in GPU virtualization that has transpired during the work of the thesis.

2.6.1 Virtual GPU Design Paradigms

Micah Dowty and Jeremy Sugerman from VMware outline different performance metrics and design paradigms for virtual GPUs[14]. Although these are design paradigms for virtual GPUs, and not GPGPU technology, they are still relevant to our design discussion of Vocale because the GPGPU facilities are embedded in physical GPUs.

Two ultimate design paradigms are established; front and back end virtualization. Front end virtualization encompass design paradigms where the hypervisor makes use of vendor provided drivers / APIs resident in the host environment to provide GPU acceleration is some form to the VM. Two main front end techniques are outlined:

- API / Driver remoting. The hypervisor emulates a virtual API / Driver for the VM's OS, in which access to the physical GPU is mediated through.

Front End	API Remoting Device Emulation
Back End	Fixed Pass-Through Mediated Pass-Through

Table 2.2: The main design paradigms behind front and back end GPU virtualization.

- Device emulation. The hypervisor emulates a GPU in the VM. The VM OS installs a driver for the virtual GPU, which is backed by whatever resources is resident in the host environment.

Back end virtualization techniques lay the virtualization boundary between the physical GPU and the graphics driver. The VM OS is exposed directly to an emulated GPU or a GPU driver. The main back end techniques are defined as follows:

- Fixed pass-through. The dedication of an entire physical GPU to the VM.
- Multiplexed pass-through. Multiplexing access to a physical GPU between VMs.

These design paradigms and their advantages / disadvantages are outlined further in Chapter 3.

2.6.2 GViM

GPU-accelerated Virtual Machines[17] provides VMs with GPU acceleration using, like our implementation of Vocale, API remoting of the CUDA API. GViM uses Xen as its base hypervisor, implementing fast device memory transfers and a scheduling policy for CUDA applications inside VMs.

As we will see in Chapter 5, device transfer bandwidth between the device and the guest is crucial to the performance of systems like GViM and Vocale. GViM proposes several ways to perform data transfers between guest and device. In CUDA, there is support for two different ways to allocate host memory. The first is the normal `malloc(..)` function, the other is the `cudaMallocHost(..)` provided by CUDA. GViM outlines three memory copy alternatives:

- The slowest alternative involves two copy operations. When the VM is started, a memory area is shared between the hypervisor and the guest's kernel space. The first copy is from the guest CUDA application's virtual address space to the guest's hypervisor-shared kernel memory. The next is the copy operation from the hypervisor-shared memory in the host to the GPU. Data is copied using *XenStore*. Note that the Xen Wiki states that the XenStore should not be used for large data transfers[30].

- The next option proposes that guest CUDA applications use `cudaMallocHost(..)` to map the application's virtual memory into the guest's hypervisor-shared kernel memory. This operation eliminates the need to copy data into the guest's kernel memory and involves only one copy of data through to the GPU.

- The last strategy to accelerate device memory transfers is to allocate the guest-shared hypervisor memory with `cudaMallocHost`, which attempts to pin the memory in place.

GViM is thus a paravirtualization approach, as to get the best bandwidth to and from the device, the programmer must use `cudaMallocHost(..)` to allocate host memory. The disadvantage of this approach is that not all programmers use this function, but rather the standard function `malloc(..)`. Further, the third approach consumes a lot of the system's main memory, because the memory is pinned and cannot be swapped out to an external storage media (refer to Section 2.5).

GViM's approaches to these problems are undoubtedly the fastest possible. The most optimal way to perform large memory transfers in and out of VMs are *zero copy* transfers, that is, there is no copying involved in passing data between the VM and its hypervisor. If one in addition can disregard physical memory fragmentation and the standard copies between kernel and user space, one will have eliminated almost all overhead incurred in these transfers.

In Vocale, we want to experiment with ways to perform memory transfers without having to resort to paravirtualization. This way, the developer does not need to rely on `cudaMallocHost(..)` to achieve acceptable device memory transfer performance.

2.6.3 New Research and Development

During the work of this thesis, some new approaches to GPU virtualization have surfaced. Investigating these is out of scope for the thesis, but it is interesting to see the direction of the development, which is sharing of PCI-Express devices among machines and dedicating hardware to VMs.

Xen Server 6, a server hypervisor released 8th March 2012, supports the dedication of physical GPUs to VMs. Of course, no effort is done to multiplex access to the GPU, so it is locked to the VM that uses it. There has also been development in Qemu, where developers are working on an OpenGL pass-through mechanism. OpenGL is a graphics library, but this work is similar to what Vocale does.

There have also been development on the hardware layer. PCI-SIG, the PCI Special Interest Group, has come up with a new standard for sharing of PCI-Express devices[16]. This is a very interesting development as there has not been hardware support for sharing hardware devices before.

This concludes our background chapter. We will now go into the design discussion and architecture of Vocale.

Chapter 3

Vocale - Design and Architecture

Vocale is an acronym for Virtual CUDA Library Extension. It is a composition of programs, libraries and drivers built on top of the Qemu hypervisor, designed and implemented as a part of this thesis. Its main goal is to provide applications in VMs with GPU acceleration, which is done by virtualizing the CUDA library. This chapter outlines the design discussion and high-level architecture of Vocale.

Vocale is built in two stages, 1.x and 2.x. The difference in these versions is how they implement data transfers between the host and the guest; other than that, they keep the same modular structure. The reason for re-implementing data transfers is that Vocale 1.x's transfer bandwidth is very slow. Vocale 1.x also causes a bug in the VM that halts the virtual CPU (refer to Section 5.2.2 and 5.3.1 for details). This chapter does not explain the difference between the versions, but their common design and software architecture.

The chapter layout is as follows. The first section in this chapter presents the overall design goals of Vocale.

The following section discusses the design choices made for Vocale. These include where the virtualization boundary is laid, which hypervisor is used for development and the OS which will be run in the host and the guest.

The third and final section introduces the modular architecture of Vocale, where each of the building blocks of Vocale and their interfaces are introduced. We will also have a look at the most central data structures in Vocale. This leads up to the implementation specific details in Chapter 4.

3.1 Design Goals

There were several design goals with the development of Vocale. These merge with the goals of the thesis (see Section 1.1) and are focused more on implementation evaluation and testing than the perfection of the system. The main design goals are as follows:

- To enable VMs with GPU acceleration in some form of a virtual GPU. The applications running in the VM need to be able to benefit from GPU acceleration (assuming that the GPU resource is backed by the host).
- Efficiency. In order for systems like Vocale to be competitive it needs to meet the near native performance standards set by the original system.
- Program modularity. Vocale and its components may change in the future; thus it needs well defined interfaces between its various components. It is also important to be able to test individual modules independently if necessary.
- Hypervisor flexibility. Vocale will be built partly in a hypervisor, and depends on the facilities provided by it in order to meet the design goals. The hypervisor needs to provide a rich set of tools that can be used to experiment with the implementation.

With the design goals explained, we can now start discussing the most important design choices of Vocale.

3.2 Design Discussion

This section discusses the design choices made for Vocale with respect to the design goals which have been defined in the previous section. In summary, we want to decide the following:

- What do we want to virtualize to bring GPU acceleration to VMs? We call this the virtualization boundary. We review VMware's design paradigms (see Section 2.6.1) for virtual GPUs.
- Platform specific details, like the OS to use for development.

- Hypervisor platform. There are several hypervisors available, and we wish to evaluate which one suits our needs best.

3.2.1 Virtualization Boundary

In order to provide GPU acceleration to a VM, the initial design choice of Vocale is to define the virtualization boundary. This boundary represents the interface in which virtualization takes place. Two main design paradigms are defined by VMware that we can use in this discussion: Front and back end. For example, back end paradigms target virtualization of physical GPUs and drivers, while front end paradigms target drivers and software libraries / APIs.

Fixed Pass-Through

Let us consider the back end paradigms first, starting with fixed pass-through. This approach dedicates a physical GPU to the VM: The host will not attempt to claim it for itself, but will allow the VM to control it. The point of this is that the VM can install any vendor provided drivers and libraries to control the device; see Figure 3.1 on the next page.

Back end virtualization may seem like a correct approach. The GPU is a hardware device, and thus it should be emulated as hardware. It gives OSs the opportunity to benefit from it, as OSs cannot directly use software libraries. It is also the only way to provide a full virtualization solution (refer to Section 2.2.2).

Implementing and maintaining this, however, is not trivial. Modern GPUs use the PCI-Express bus to interface with the processor, so we need to emulate this high-speed bus in the VM. The PCI-E specification also costs money, and we may need this if there is no framework for virtual PCI-Express devices in existing hypervisors.

While we do not know any details about PCI-E, we can still imagine problems related to this task. An issue that will arise in this scenario is when one of the GPU accelerated processes in a VM attempts to perform a device memory transfer. The physical GPU can typically perform DMA in the physical memory of the *host*. The GPU driver which is running in the guest, however, will create DMA mappings from virtual memory to the physical memory of the *guest*. The problem is that these DMA mappings hold no correspondence with the physical memory of the host, so when the

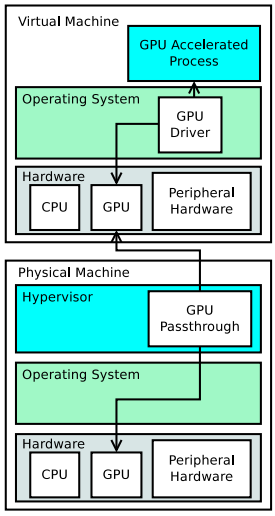

Figure 3.1: Shows how applications in a VM can be GPU accelerated using fixed pass-through.

device performs DMA with these mappings it will read or write rubbish. A solution to this could be to mirror the physical memory layout of the host with the physical memory layout of the guest, but this is not trivial to do.

Dedicating a single GPU to a VM also does not give us any opportunity to share the GPU resource between a host and its guests. This is important to for example computing clusters. Therefore we do not consider fixed pass-through as a viable alternative for our virtualization boundary.

Multiplexed Pass-Through

Multiplexed GPU access is another back end paradigm. Its purpose is to provide an opportunity to share GPU resources at a physical level. This requires knowledge of physical hardware that is entirely closed to the public by the GPU vendor. GPU hardware is meant to be used by one machine only, and multiplexing access to one may prove an extremely complex task. Aside from this, it suffers from the same problems as fixed pass-through as well. Thus we also consider multiplexed access a no-go for the virtualization boundary of Vocale.

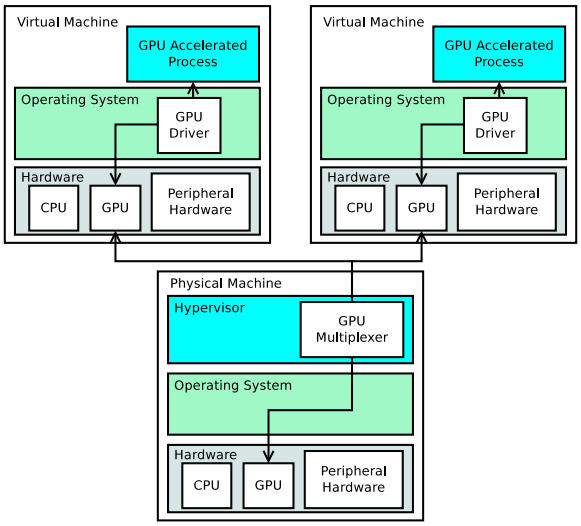

Figure 3.2: Shows how applications in a VM can be GPU accelerated using multiplexed pass-through.

Device Emulation

The concept of device emulation lies on a boundary between front and back end paradigms. VMware's hypervisors[14] make use of this solution. Device emulation presents a virtual GPU to the VM that will never change in the way that proprietary GPUs do, in that it exists only in software. This offers a more relaxed environment where the challenge is to synthesize GPU operations using whatever resources are available in the host. This option brings about more options, however, and some are more related to front end paradigms than the others. To bring GPU acceleration to the VM, one could for example do either of the following:

- Emulate a device that fools the proprietary GPU driver, for example the NVIDIA driver, into believing it is running on an NVIDIA GPU. This alternative faces the same problems of closed architecture.
- Make a new, open source implementation of a GPGPU library such as CUDA or OpenCL that fits with the emulated device. This, however, would prove a very large task.

While we consider device emulation more interesting, our proposals above either involves massive amounts of work or faces the same problems as back end paradigms. Our final proposal is a pure front end technique called remote procedure calling.

Remote Procedure Calling

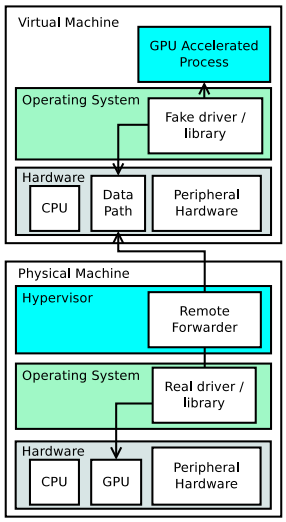

Figure 3.3: Shows how applications in a VM can be GPU accelerated using remote procedure calling.

Front end techniques in general are much more interesting for Vocale, because they leave the hardware layer behind and targets direct virtualization of drivers and libraries. While drivers often suffer from proprietary details, software libraries generally have some kind of API that is known to the public. Knowing the API of the library or driver, it is possible to implement a fake version of it and remotely call the *real* interface in the host (see Figure 3.3).

For example, if an existing library has two functions, `remote(..)` and `gpu(..)`, a virtual library implementation in the guest would implement those two and *forward* them to the real one in the host. The calls and callbacks are sent across the guest / host boundary to service requests from the VM's user space.

Using remote procedure calling, details and internal workings of the driver

/ library can be left out to a large degree, owing to the fact that we only forward and return function calls. To the contrary of back end virtualization, these systems have the potential for easy maintenance as the vendor itself will manage hardware support and backward compatibility. The only concern with these solutions is that, depending on the library functionality, some knowledge of the internal workings of the library may be necessary. Note that these solutions center around a paravirtualized approach. This is because the virtual library or driver needs to know that it is running in a virtual environment to operate.

Because of this, Vocale implements GPU acceleration by the front end design paradigm, remote procedure calling, of a GPGPU library. Refer to Table 3.1 on the next page and 3.2 on page 42 for a summary of the points above.

Now that we have decided to virtualize a GPGPU library using remote procedure calling, the next question is what library we will emulate, and on what OS platform.

3.2.2 OS Platform and Library

In this section, we will discuss which GPGPU library we will emulate in Vocale. We also decide which OS platform we want to base the implementation on.

GPU acceleration is provided through two commonly known libraries, OpenCL and CUDA. We will focus on emulating CUDA, as both OpenCL and CUDA are similar in architecture, and the writer is more familiar with the latter.

Since we are to implement our own, virtual, CUDA library, we need to establish some minimum design goals for it as well.

- Developers should be able to invoke `nvcc` (refer to Section 2.3.1) as usual to compile and link CUDA applications.
- Function calls must be working (refer to Section 2.3.1).
- The guest must be able to execute kernels.
- The guest must be able to perform device memory transfers.

The project needs to handle two OSs; one for the host and one for the guest. Our implementation will involve virtual hardware, and that means

	Advantages	Disadvantages
Fixed Passthrough	- Full virtualization. - Vendor handles drivers and libraries.	- Need to know PCI-Express details. - Need to know GPU specific hardware details. - No GPU sharing. - DMA may be hard to achieve because of the VM's MMU. - Proprietary, ever changing GPU architectures.
Multiplexed Passthrough	- Full virtualization. - Vendor handles drivers and libraries. - Shared GPU resource.	- Same disadvantages as for fixed pass-through. - GPUs are not meant to be shared in hardware.

Table 3.1: Overview of advantages and disadvantages of back end design paradigms.

	Advantages	Disadvantages
Device Emulation	- Full / Partial virtualization. - A single GPU that rarely changes.	- Synthesizing virtual GPU operations with whatever resources resident in the host. - High complexity of the task in general. - Proprietary details may still be necessary.
API Remoting	- Paravirtualization. - Vendor handles backwards compatibility and maintenance. - Forwarding API calls should not prove difficult, as the API is open.	- May require insight into how the library works.

Table 3.2: Overview of advantages and disadvantages of different front end design paradigms.

working with device drivers. Using Linux provides us with free hands to experiment. We want to use the 64 bit version of Ubuntu in both the host and the guest, which helps coping with architecture difficulties that can occur between for example 32 - and 64 - bit systems and systems with different byte-order endianness.

3.2.3 Hypervisor Platform

We have chosen to emulate CUDA in a VM to provide the VM's applications with GPU acceleration. In the last subsection, we determined to use the Ubuntu Linux distribution as our OS of choice for the guest and the host.

The last part of the design discussion revolves around the choice of hypervisor. The hypervisor is an important part of the design because function calls and callbacks between the host and the guest will take place through it. In other words, part of the implementation will be an extension to the hypervisor to allow for host - guest communication.

Unfortunately, both proprietary and open source hypervisors are not very well documented, and information about them has been hard to find. As the Qemu developers say, "only the source code tells the full story"[24]. Also, we have experienced that information often is out of date, and as we have said earlier there has been research and development in our field during the work of the thesis.

There exists some work that attempts to outline the main differences between them[20], and we will attempt to make use of this. The following list summarizes our criteria for Vocale's base hypervisor.

- The hypervisor is active, that is, under constant development and backed by a developer community.
- Linux support, following our discussion in the previous subsection.
- Open source and as good documentation as possible.
- A type 2 hypervisor (refer to Section 2.2.3) for easier debugging and system architecture; we believe it is hard to debug a program directly on hardware.
- Not been the target of previous work in the area of remote procedure calling of the CUDA library.

- A rich set of tools to facilitate data transfers between the guest and the host.

We will now argument for our choice of hypervisor.

Xen is a popular, fast and reliable; however it is also a type 2 hypervisor. The need to modify guest OSs depending on the virtualization capabilities of the hardware could pose a problem if we want to test the implementation on different OSs. Despite this, Xen represents an opportunity to share computational resources. Unfortunately, the GViM project[17] has already implemented CUDA forwarding in this hypervisor.

VMware hosts a number of virtualization solutions, including one open source type 1 hypervisor named VMware Player. VMware Player is claimed to achieve great performance, but we could not find much documentation in terms of internal functionality and design.

VirtualBox is a Type 1 hypervisor for both Windows and Linux and doesn't really have any disadvantages as long as the host computer supports virtualization in hardware. This makes it a viable option for implementation of GPU virtualization.

Qemu is an open source Type 1 hypervisor for Linux. It is documented to some degree and suitable for self-study, with a supporting developer community behind it. We have also noted that it is used by other hypervisors, like Xen, to implement virtual hardware. This leads us to believe that it has a good internal framework for virtual hardware. Since it is made for Linux, experimenting with for example DMA is easier than on for example Windows. Qemu also supports KVM which is a kernel module that boosts the performance of the hypervisor.

As we have said, finding good, up to date information about these hypervisors has been hard. One has to actually study the source code to be able to find detailed information. We believe that Qemu will be able to suit our needs. This is mainly due to the fact that researchers state that its hardware model is being used by other hypervisors. We think Xen would also be a good choice, as it is used for server virtualization and relevant for cloud computing. But since Xen has already been the target of GPU virtulization in the GViM project, we choose to work with Qemu.

With our high level design choices made, we can start outlining Vocale's software architecture.

3.3 Vocale - Architecture

As we have stated in the beginning of this chapter, Vocale is built in two iterations. Regardless of version, however, Vocale keeps an identical, modular structure to accommodate change. This is one of Vocale's design goals. Figure 3.4 on the next page shows all the main components of Vocale in a running system. Starting from top to bottom, we now describe each of them.

CUDA Application. A normal CUDA process running in the guest OS. It runs and operates just like it would in a real, non-virtual CUDA environment (full virtualization).

Virtual CUDA library. Our implementation of the CUDA library. Its main purpose is to forward function calls and kernel launches to the CUDA forwarder described below. CUDA code can compile and link as normal towards this library by invocation of `nvcc` (refer to Section 2.3.1).

Transfer Library. A shared library implementing a simple interface for transferring data to and from the host.

Data Path. The data path provides a line of communication between the host and the guest. It is always implemented as virtual Input / Output (I/O) hardware, as this is the natural way for a CPU to communicate with the outside world. The data path is complex in sense that it requires an implementation in both the host (Qemu) and the guest OS (a driver). For example, virtual hardware like a PCI device needs a driver in the guest OS as well as an implementation in Qemu.

CUDA forwarder. The CUDA forwarder is responsible for forwarding function calls, data and kernel launches to the real CUDA library. It also passes back any results, also called the callback, to the guest.

CUDA Library. This is the real, proprietary CUDA library. The forwarder links to it like a normal CUDA application to accomplish its tasks.

CUDA Driver. The real, proprietary CUDA driver. It is used by the CUDA library to communicate with the physical GPU.

Vocale can be split in three parts following the list above.

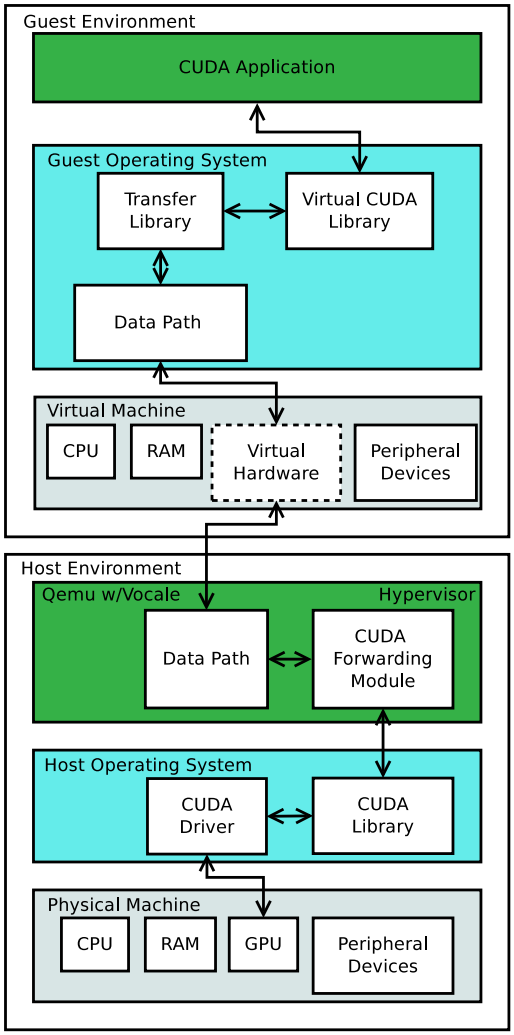

Figure 3.4: The general architecture of Vocale.

1. The virtual CUDA library.
2. The data path.
3. The CUDA forwarder.

The following subsections describe their architecture in closer detail, and leads up to the implementation specific details in the next chapter.

3.3.1 Virtual CUDA Library

The virtual CUDA library imposes the role of the real one as provided by NVIDIA. When installed in the guest OS, it will enable guest CUDA applications to use the host's GPU for computation. The virtual library should be a full virtualization system, that is, its users should be able to use it without requiring any knowledge they are running on a virtual system (refer to Section 2.2.2). CUDA API calls and kernel launches should be intercepted by the virtual library and sent to the CUDA forwarder (described further on in Section 3.3.3). The forwarder executes the call or kernel launch and passes back the result.

All references to the CUDA API is done through two NVIDIA provided headers: `cuda.h` and `cuda_runtime.h`. These must also be included by the virtual library to implement the relevant API functions declared in the headers. It is compiled and installed as a shared library (refer to Section 2.4.1).

It is not required, nor desirable, to override all the API functions and mechanisms of CUDA. For example, we do not want to override kernel code compilation (refer to Section 2.3.1) or device specific math functions. The virtual library should only override the functionality that requires the presence of a physical GPU. This means that the proprietary CUDA library must be installed alongside the virtual library, overriding only the required functionality and letting its real counterpart do the rest. Fortunately, Linux's shared library implementation is designed just for these purposes, and it is easy to override a selection of functions from a library.

Function call forwarding and result callback is done by passing a data structure containing any input and/or output parameters of the desired function along with a call header. This data structure can be seen in code example 3.1 on the following page. The following list describes its fields.

```
struct callHeader{
    struct header   head;
    int             callType;
    int             callID;

    cudaError_t     respError;
    CUresult        drvRespErr;
};

struct cudaGetDevicePropertiesStruct{
    struct callHeader callheader;

    struct cudaDeviceProp prop;
    int device;
};
```

Code Example 3.1: A sample data structure for an API function's input / output parameters and call header.

header. The header is a nested data structure that can contain request type identifiers. The only identifier used in Vocale is the one for call forwarding requests, so this field can be ignored.

callType. This member was intended as a second level classification, for example kernel launch, memory transfer, normal call et cetra. Our work with Vocale showed us that this field is not needed, as all the operations of the virtual library can be solved using remote procedure calling. There is no need to specify a kernel launch, memory transfer et cetra.

callID. The ID corresponding to the requested call. There is an ID for each API function that Vocale can forward.

respError / drvRespErr. Error return types. Nearly all CUDA calls return these depending on which API they belong to (refer to Section 2.3.1). The return value is always passed back this way.

Code example 3.1 also depicts the function parameters of the `cudaGetDeviceProperties(..)` function, which retrieves device specific properties of a selected device installed on the system. It takes an input parameter, the *device number*, and returns the *properties* of the device. Thus input and output parameters are carried in both directions of the call request. The call header

can be seen as the first field. All function calls are forwarded and returned in this way.

With the design of the virtual CUDA library explained, we can now continue with the data path, which is used to transfer data between the guest and the host.

3.3.2 Data Path

The data path is the most central system of Vocale, in that it implements communication between the virtual library and the forwarder. This subsection describes the message passing data structure, information flow and the interfaces that the data path must implement in the VM and the hypervisor.

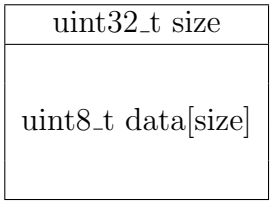

Figure 3.5: Data transfer message structure.

Any message passed through the system has the message structure seen in Figure 3.5. The data path is not concerned with the contents of the message, just the size and the actual data. Note the use of standardized sizes to ensure correct interpretation between different guest and host CPU architectures.

Our data path is to be used by the guest to request forwarding of function calls, kernel launches and device memory transfers from the host. Thus we assume that any communication is initiated from the guest; the host will not have any way to pass data directly to the guest.

The *transfer library* block in Figure 3.4 on page 46 is the guest's interface to the data path. Its interfaces are shown in code example 3.2 on the following page and are defined as follows.

init() is an initialization routine that is called whenever a new guest CUDA application using the virtual CUDA library starts. What it does depends on the implementation of the data path. It is called before main by marking it as a constructor as shown in code example 3.3 on the next page.

```
int init(void)

int sendMessage(
    void* msg_buf,
    size_t msg_sz
    )

size_t recvMessage(
    void** msg_buf
    )
```

Code Example 3.2: Interface provided by the transfer library to communicate with the host.

```
void __attribute__((constructor)) init(void){
    /* Initialize */
}
```

Code Example 3.3: A program initialization function called before main(..).

sendMessage(..) sends `msg_sz` bytes from the `msg_buf` buffer through the data path. It returns the number of bytes written or a negative error value on failure.

recvMessage(..) receives data from the data path, allocating and storing them in a buffer `*msg_buf`. It returns the length of the buffer in bytes or a negative error value on failure.

The point of having this interface is to be able to make changes to the data path without having to redesign Vocale. Changes are not necessary as long as they keep this interface intact, and it is possible to extend it later if needed.

The CUDA forwarding module in Figure 3.4 on page 46 also needs an interface to the data path. Therefore, the data path provides a *message handler registration function* to which the forwarder registers a *message handler function*. These can be seen in code example 3.4 on the next page. The data path implementation calls the message handler when messages are received.

The following list describes the parameters of the message handler code example 3.4 on the following page.

msg is a pointer to the received message.

```
typedef void (messageHandler)(
    void* msg,
    uint32_t msg_sz,
    void** respMsg,
    uint32_t *respMsgSz
    )

void registerPortHandler(
    IOListenerPortHandler *handler
    );
```

Code Example 3.4: The host side interface of the data path.

msg_sz is the size of the received message (in bytes). It should always be checked before accessing `msg`, as a message can have size 0.

respMsg is a double pointer to a buffer. The message handler can choose to allocate a buffer in `*respMsg` where any response message is stored. In that case, the handler must also set `*respMsgSz`.

respMsgSz is a pointer to the size of the response message. If this is not set, but `*respMsg` is allocated with a response message, a memory leak will be the result.

3.3.3 CUDA Forwarder

The purpose of the forwarder is to handle any function or kernel launch request made from the guest. It resides within the Qemu process, and links with the real CUDA library like a normal CUDA application.

The CUDA forwarder must implement a handler function and register it when Qemu is started, as according to code example 3.4. When messages are received, the forwarder looks up the call header shown in code example 3.1 on page 48, and assumes the layout of the message by determining the function call ID. The function is executed with any input / output parameters and the results are sent back to the guest using the message handler parameters.

There is not much to say in general about the forwarder, except that it blindly forwards requests from the guest. A more detailed look at the implementation is given in Section 4.2.3.

We have now outlined the design of Vocale, and we are ready for the implementation specific details in the next chapter.

Chapter 4

Vocale - Implementation

This chapter describes the implementation specific details of Vocale 1.x and 2.x. As we stated at the start of the previous chapter, these differ in that Vocale 2.x implements a new data path. In this chapter we will see how.

Our development process of Vocale represents a significant amount of work. It involved understanding Qemu's software architecture, kernel / library development and last but not least, internal properties of CUDA. Much of this is covered in this rather than the background chapter, because it is very extensive and was a direct part of our implementation work.

In this chapter we reference Vocale's source code (refer to Section 1.3). The main folders of the source tree can be seen in Figure 4.1 on the next page. Depending on the context, source code references are made to the `vocale-1.x` and `vocale-2.x` folders. These keep the same internal structure:

src is the folder that contains the Qemu source code. It has been modified to build with Vocale's extensions.

lib is a folder used by the guest. It contains the virtual CUDA library source code.

driver is also used by the guest. It contains the drivers used by Vocale to communicate with the host.

The source tree also contains a `Python` folder. This folder contains scripts to auto generate code, as described at the end of Section 4.2.2.

The chapter layout is as follows. The first section describes the Qemu object model, which is a pseudo object-oriented programming model for Qemu

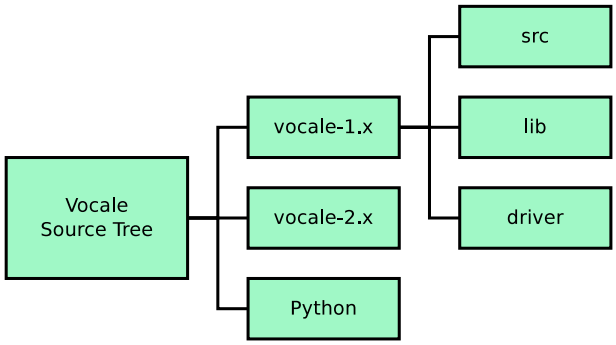

Figure 4.1: The source tree structure of Vocale.

in C. Adding virtual hardware should always be done using this model, which supports both inheritance and interfaces. Vocale 2.x uses this model to implement faster device memory transfers - Vocale 1.x does not.

The second and third sections outline the implementation of Vocale 1.x and 2.x, respectively. The second section is more detailed as it has to describe the implementation of all the design components depicted and described in Section 3.3. The fourth is shorter in that it only has to introduce the new way to perform data transfers and a small patch in the forwarder.

4.1 Qemu's Object Model

Qemu provides an object model to add code and components to Qemu, for example virtual hardware. It is a pseudo object-oriented architecture written in C that supports the following functionality:

- Creating new classes and object types.
- Class and object initialization and finalization (constructors and destructors).
- Inheritance from other classes.
- Interfaces.

These are many of the benefits of working with high level object oriented languages such as Java, C++ and C#, but the object model is more complex

```
qemu-system-x86_64 -device virtio-serial
```

Code Example 4.1: Virtual device initialization from the command line.

```
#define type_init(function)                          \
        module_init(function, MODULE_INIT_QOM)
```

Code Example 4.2: Sample macro for registration of new class types.

as it is written in C. It is described in `include/qemu/object.h` under the `src` folder, but as Vocale 2.x uses it we will also give an introduction to it here.

Code example 4.1 shows how a virtual hardware device, `virtio-serial`, is initiated from the command line. `virtio-serial` is a virtual serial device that is implemented as a class in Qemu's object model, and instances of the class are initialized in this manner. Any class type can implement options in the form of a comma separated list after the name of the device; refer to [1] for examples.

We now take a look at how new class types can be added to Qemu. Qemu provides a set of macros that are used to register new object types in `src/module.h`. Code example 4.2 shows one of these macros.

Simply put, if that is indeed possible, this macro registers an initialization function with Qemu whose job is it to register type information about the new class. To do this work, the initialization function makes use of a type registration function, `type_register_static(..)`, and type data structure, `struct TypeInfo`, shown in code example 4.3 on the following page. The type data structure fields are explained below.

name is the name of the class used for initializing instances of the class according to code example 4.1.

parent is the name of the parent class from which the new class inherits attributes and functions[1].

instance_size it the size of the data structure representing an object of a class. This data structure always embeds its parent class' object data structure.

[1]Like high level languages such as Java, all objects in Qemu are children of a basic object type.

```
Type type_register_static(const TypeInfo *info);

struct TypeInfo
{
    /* Some members are omitted for simplicity */

    const char *name;
    const char *parent;

    size_t instance_size;
    void (*instance_init)(Object *obj);
    void (*instance_finalize)(Object *obj);

    size_t class_size;
    void (*class_init)(ObjectClass *klass, void *data);
    void (*class_finalize)(ObjectClass *klass, void *data);
};
```

Code Example 4.3: The data structure used to register information about new object types.

instance_init and instance_finalize are function pointers to the instance constructor and destructor of a class. These are called when new objects are created and killed.

class_size is the size of the data structure representing a class. This data structure always embeds its parent class' class data structure.

class_init and class_finalize are function pointers to the class constructor and destructor.

The job involved in creating new object types is thus to implement all the above functions. In addition to initializing itself, it is the constructors' responsibility to override any inherited functions from parental classes and objects. Examples of this can be seen in Section 4.3.

We will get back to the Qemu object model in Section 4.3, but first we will describe the implementation of Vocale 1.x.

4.2 Vocale 1.x

The initial implementation of Vocale is based on Qemu version 0.15.x. All file references are to the `vocale-1.x` folder of the source tree as described in

the beginning of this chapter. This section goes into implementation specific details of Vocale 1.x according to the architecture outlined in Section 3.3.

Vocale 1.x implements the virtual CUDA library, a data path using I/O ports and the CUDA forwarder in Qemu. Figure 4.2 on the following page shows an overview of the implementation, which is very similar to the figure given in Section 3.3 on page 46.

The next subsections describe the three components that Vocale 1.x implements.

1. The data path. It consists of the `IOlistener Driver`, the `IOlistener Virtual Device` and the `Transfer Library` blocks in Figure 4.2 on the next page. We want to describe this component first because it is central to the two others.

2. The virtual CUDA library. It consists of the `Virtual CUDA Library` block in Figure 4.2 on the following page. We describe how it captures CUDA API calls and kernel launches.

3. Finally, the `CUDA Forwarding Module` is explained. The forwarder executes CUDA calls and kernel launches natively on the real CUDA library.

4.2.1 Data Path

One of the challenges of Vocale is to be able to pass data between the host and the guest. To do this it is necessary to exploit the natural ways in which a computer exchanges data with the outside world: In other words, hardware. Modern computers host GPUs, USB controllers and network devices, all of which are able to transmit data in and out of the computer.

Our requirement is to be able to pass data from the guest CPU's main memory to the main memory of the host. At a low level, one might say that all memory used by the guest and the host reside in the same physical memory (unless it is being swapped out). The reason we cannot directly share this memory is because of the MMU of modern processors (refer to Section 2.5). The MMU creates several distinct address spaces. One for each process and OS in the guest and the host. We cannot directly pass memory between these worlds, because memory pointers in one address space has no meaning in another (remember Figure 2.9 on page 27). So the only solution

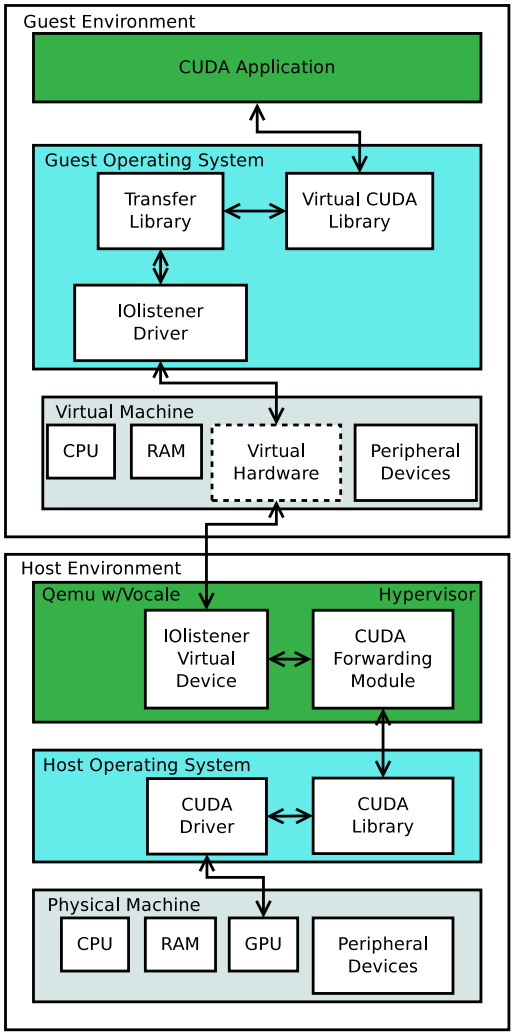

Figure 4.2: The implementation specific components of Vocale 1.x.

is to copy data between the host and the guest using some shared interface; and then we are back at hardware.

Our data path implementation uses I/O ports to pass data between the host and the guest. The definition of I/O ports varies across processor architectures, but for Intel processors it can be viewed as a separate address space for hardware[5, Chapter 9]. The guest processor can read and write data to our I/O port to exchange data with the host environment.

It is important to note that the I/O port communication can only be triggered from the guest processor using assembly instructions like `inb` and `outb`. Hence data transfers can only be triggered when the guest CPU reads or writes the I/O port.

The implementation is split in two parts.

- On the host side, Vocale implements a form of virtual hardware device called the `IOlistener` in Qemu. It is not implemented with Qemu's object model. This device can send and receive data on a single I/O port.
- On the guest side, Vocale implements a Linux character device driver for the `IOlistener`. The driver, or kernel module, provides a simple interface that can be used to exchange data with the `IOlistener` forwarder in the host.

We will now describe these two interfaces.

Host Side: IOlistener

The `IOlistener` is a form of virtual hardware device integrated into Qemu; but it does not use Qemu's object model. Its source code can be found in `src/iolistener.c/.h`.

This component is able to respond to I/O port reads and writes from the guest CPU by registering itself to an I/O port. An API to do this is provided by Qemu through the `ioport.h` header file, as can be seen in code example 4.4 on the following page. The following list describes the parameters of the functions.

func is a handler function that gets called when the specified port numbers are accessed.

```
int register_ioport_read(pio_addr_t start, int length, int size,
    IOPortReadFunc *func, void *opaque);

int register_ioport_write(pio_addr_t start, int length, int size,
    IOPortWriteFunc *func, void *opaque);
```

Code Example 4.4: I/O port interface provided by Qemu.

```
typedef void (IOListenerPortHandler)(
    void* msg,
    uint32_t msg_sz,
    void** respMsg,
    uint32_t *respMsgSz
    )

struct IOlistener_common* registerPortHandler(
    IOListenerPortHandler *handler,
    pio_addr_t portstart,
    pio_addr_t control_port);
```

Code Example 4.5: Above: The message handler function type. Below: A function to register handlers to a port.

start is the first port number to register with. The `func` handler will be called when the guest CPU reads or writes this port.

length is the number of consecutive ports after `start` to which `func` will be called on read or write access.

size is the size (in bytes) of each access. Valid values are 1, 2 and 4. For example, reading a port which has been set up with a size of 4 will return a number with a length of 4 bytes.

The `IOlistener` implements the message handler registration function as shown in code example 3.4 on page 51; the implementation can be seen in code example 4.5. Note that the names of the functions have changed. The `control_port` is unused.

We will explain how the `IOlistener` communicates with the guest after the next subsection, but we will not show any more code examples on this. The source code is too large, so readers are referenced directly to the source code. We will not explain the `IOlistener`'s device driver.

```
virtual@virtual-machine:/dev$ ls -lha
drwxr-xr-x  15 root root         3.4K 2012-04-24 15:42 .
drwxr-xr-x  22 root root         4.0K 2011-11-29 16:51 ..
crw-------   1 root root      10, 235 2012-04-24 15:42 autofs
crw-------   1 root root       5,   1 2012-04-24 15:42 console
crw-------   1 root root      10,  58 2012-04-24 15:42 cpu_dma_latency
crwxrwxrwx   1 root root     251,   0 2012-04-24 15:42 cudamodule0
crw-------   1 root root      10,  61 2012-04-24 15:42 ecryptfs
```

Code Example 4.6: `ls` output listing for a VM running Vocale 1.x.

Guest Side: Kernel Module and Transfer Library

The virtual CUDA library needs to be able to communicate to the forwarder. This is done through port I/O, which is not a normal user space operation and therefore requires kernel privileges[2]. Therefore, the guest is extended with a character device driver (refer to Section 2.4.2) and a simple transfer library that communicates with it. The transfer library provides the interface shown in example 3.2 on page 50 to the virtual CUDA library.

The source of the kernel module can be found in `driver/cudamodule.c`. Note that the name is slightly misleading, as the driver has nothing to do with CUDA. Further, we do not want to delve too much into implementation specific details on our driver. This is because it will take up too much space. Readers are referenced to the source code and Linux device driver literature[5, Chapter 3] for the full story. We do, however, want to show how the driver provides a way to read and write data between the guest and the host.

As described in Section 2.4.2, character drivers are represented as special files in Linux's file system. Example 4.6 is an `ls` output listing from a VM running Vocale 1.x. It shows several character drivers, among them, `cudamodule0`. This is the file that represents our kernel module.

The point of representing drivers as special files in Linux's file system is to allow simple communication between user and kernel space. A character driver can be opened, read and written just like a normal file[3]. All character drivers must implement a set of functions that are called when a user space application opens, closes, reads or writes the module. We now show how our kernel module implements the two latter ones. Note that the examples have

[2]There may be workarounds to allow port access from user space[5], but this was never implemented.

[3]It can also be communicated with using special device commands called *ioctl* commands, but we have not implemented this.

```
static ssize_t cudamod_write(
                        const char __user *user_buf,
                        size_t size){

    /* Write the message size */
    for(i = 0; i < 4; i++){

        outb(msg_size[i], IOLISTENER_IO_DATA_START);
    }

    /* Copy user space buffer into kernel memory */
    kern_buf = vmalloc(size);
    if(copy_from_user(kern_buf, user_buf, size) != 0)
        return EFAULT;

    /* Write message to IOlistener  */
    for(i = 0; i < size; i++){

        outb(kern_buf[i], IOLISTENER_IO_DATA_START);
    }

    vfree(kern_buf);

    return size;
}
```

Code Example 4.7: A simplified example of the `cudamodule` write handler.

been simplified.

Code example 4.7 shows how data is written through the I/O port. This function gets called when a user space application writes data to the `cudamodule0` device file. All transfers start by writing the four bytes of the message size. Then, the user space buffer is copied into kernel memory, as port I/O access must be done through the kernel address space. Finally, the entire message is written, one byte at a time.

Code example 4.8 on the following page shows the opposite operation. Here, data of some size is read from the port. Unfortunately, the example is potentially confusing because it does not start by reading the size of the data. Instead, it just reads the specified amount of bytes. This is an implementation error that is handled by the transfer library, as we will see in a moment.

A kernel module must be loaded. We want to make a note that the `cudamodule` is loaded by the `/driver/load.sh` shell script, which can be invoked manually or from a boot command file like `/etc/rc.local`.

With the `cudamodule` read and write handlers explained, we can present the final part of the data path. The transfer library source can be found in

```
static ssize_t cudamod_read(
                        char __user *user_buf,
                        size_t size){

    /* Read in size bytes to kernel memory */
    kern_buf = vmalloc(size);

    for(i = 0; i < size; i++){

        kern_buf[i] = inb(IOLISTENER_IO_DATA_START);
    }

    /* Copy the data to user space */
    if(copy_to_user(user_buf, kern_buf, size) != 0){

        return EFAULT;
    }

    vfree(kern_buf);

    return size;
}
```

Code Example 4.8: A simplified example of the `cudamodule` read handler.

`lib/transfer.c/.h`. The implementation is fairly simple.

Code example 4.9 on the next page shows the implementation of the transfer library interface as shown in example 3.2 on page 50. The `fd` parameter in the read and write calls is a file descriptor that correspond to the `cudamodule0` device file. The following list describes the function calls.

The sendMessage(..) implementation is fairly simple. The `cudamodule` takes care of the size of the message as described above, so it can simply write the data to the driver.

The recvMessage(..) function is slightly more complex because the driver do not handle the size of the message. Instead, the library must first read in the size of the message, and then the message itself.

We can now outline the final part of the data path, which is the communication protocol between the guest and the host.

Communication Protocol

The final part of our description of the data path unravels the communication protocol between the guest and the host.

```
int sendMessage(void *msg_buf, size_t msg_sz){

    if(write(fd, msg_buf, msg_sz) < msg_sz)
        return FACUDA_ERROR;

    return msg_sz;
}

size_t recvMessage(void **msg_buf){


    uint32_t msg_sz;
    if(read(fd, &msg_sz, 4) < 0){
        return FACUDA_ERROR;
    }

    if(msg_sz == 0){
        *msg_buf = NULL;
        return 0;
    }

    *msg_buf = malloc(msg_sz);
    if(read(fd, *msg_buf, msg_sz) < msg_sz){
        return FACUDA_ERROR;
    }

    return (size_t)msg_sz;
}
```

Code Example 4.9: Transfer library implementation.

It is important to realize, as we have mentioned earlier, that all communication must be initiated from the guest. Moreover, we do not support several messages "in flight"; that is, when someone writes a message to the transfer library, they must also make the call to receive the response message before any other communication can take place.

The message protocol works as follows (we apologize for the lack of figures, there will be more illustrative examples later):

1. The transfer library gets a `sendMessage(..)` request with a pointer to user space data and a size.
2. The transfer library writes the buffer and size to the `cudamodule` device file.
3. The `cudamodule` writes the size of the message to the `IOlistener`, one byte at a time.
4. The `IOlistener` receives the size. It allocates a buffer to hold the message and prepares for the transfer.
5. The `cudamodule` starts writing the message, one byte at a time, to the `IOlistener` .
6. For each byte transferred, the `IOlistener` copies a byte to its message buffer from step 4.
7. When all the bytes of the message has been transferred, the `IOlistener` calls the message handler function, if any.
8. When the message handler is complete, the `IOlistener` prepares to return the size of the response message. It can be 0.
9. The `sendMessage(..)` function from step 1 returns. The user of the transfer library must now make to receive the response data.
10. The transfer library gets a `recvMessage(..)` request.
11. The transfer library reads four bytes from the `cudamodule` device file.
12. The `cudamodule` reads the four bytes from the I/O port. These bytes correspond to the size of the response message.

13. The transfer library gets the size of the response in these four bytes and allocates a buffer to hold the response message.

14. The transfer library makes to read the message from the `cudamodule` with the response size.

15. The `cudamodule` reads in the response message from the `IOlistener`, one byte at a time, and copies it to the transfer library.

16. The transfer library returns the message buffer to the calling user space application.

We have now explained how communication takes place between the guest and the host. Our next subsection will explain the implementation of the virtual CUDA library. We describe how it captures function calls and manages kernel launches.

4.2.2 Virtual CUDA Library

This subsection explains the implementation of the virtual CUDA library, which resides in the guest. This subsection is the most detailed in the thesis, as it represents a lot of low level work.

Applications in the guest should be able to use the virtual library according to the design goals in Section 3.2.2 on page 40. The reader should understand the main CUDA concepts described in Section 2.3.1 to fully appreciate all the points made in this subsection.

The virtual CUDA library source can be found in `/lib/virtCudaLib.cpp`. The library is easily compiled and installed by invoking `make lib` and `sudo make reinstall` in this directory. This compiles, links and installs the library as covered in Section 2.4.1.

Developers working in VMs should be able to invoke the CUDA tools, for example `nvcc`, as normal without any knowledge that they are using a virtual library. Since `nvcc` handles linking with the real CUDA library, it is necessary to "fool" it into linking object code with the virtual library *in addition* to the real one. Remember that we don't want to override absolutely everything from the CUDA library, just the functionality that requires the presence of a physical GPU (see the third paragraph of Section 3.3.1). There are two ways in which this can be done:

```
LIBRARIES        =+ -lfakecuda -lcudart
```

Code Example 4.10: A snippet of the `nvcc.profile` file.

- Setting the LD_PRELOAD environment variable. This can be used to preload the virtual library in the linking phase, effectively overriding the real one.
- Changing a special file, `/usr/local/cuda/bin/nvcc.profile`. This file contains the library names that are used for linking of CUDA applications. Example 4.10 shows a simplified snippet of the file and how it can be edited to override the standard library (`lcudart` is the real library).

The latter is the one implemented by Vocale, as the first one should not be used as a permanent solution for overriding library functionality[29]. When `nvcc` builds CUDA executables in the guest, it will first look for symbol references in the `fakecuda` library, which implements fake versions of the required functions. Everything else is linked with the standard CUDA library, `cudart`.

The rest of the subsection explains how the virtual CUDA library implements function calls, device memory transfers and kernel launches.

Function Call Forwarding

We will now outline how function calls are forwarded throughout Vocale's subsystems.

The following list illustrates the general algorithm for performing remote procedure calling from the guest (see Figure 4.3 on the next page):

1. A guest CUDA application calls a library function from the CUDA API, for example `cudaGetDeviceCount(..)`. The virtual CUDA library intercepts the call. Its implementation of `cudaGetDeviceCount(..)` constructs a call request message containing:
   - A request type identifier. Always set to `normal call`.
   - A call identifier for `cudaGetDeviceCount(..)`.
   - A data structure containing function input / output parameters.

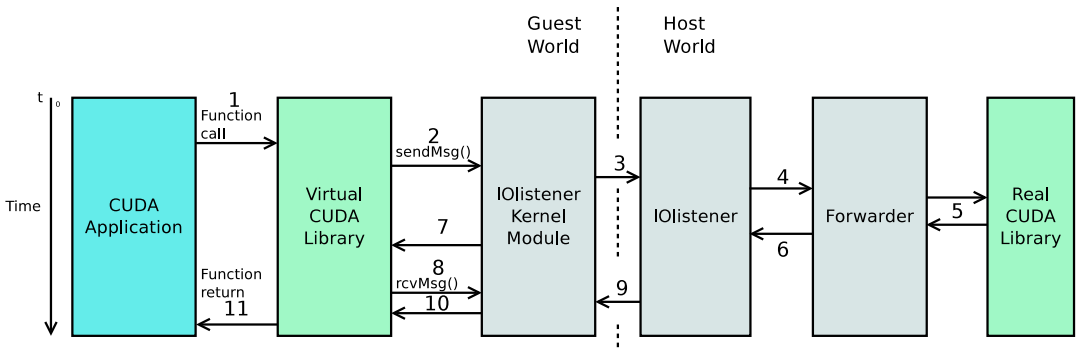

Figure 4.3: Call forwarding through Vocale's subsystems.

2. The virtual CUDA library writes the call request message to the `IOlistener` kernel module via the **transfer** library (not shown in Figure 4.3 for the sake of simplicity).

3. The `IOlistener` kernel module writes the message to the `IOlistener` through I/O ports.

4. When the message has been received, the `IOlistener` calls the forwarder with the received message through the message handler function.

5. The forwarding module in Qemu receives the message through its message handler. It identifies the request as a function call and looks up `cudaGetDeviceCount(..)`. The real function is called with the transferred input / output parameters.

6. The forwarding module constructs a reply message containing any output parameter and the result (`cudaError_t` or `CUresult`). It notifies the `IOlistener` that it wants to respond with some data by setting the return message size parameter of its message handler.

7. The virtual CUDA library gets control again.

8. The virtual CUDA library makes a request to receive data from the `IOlistener` kernel module through the transfer library.

9. The `IOlistener` kernel module reads I/O ports to fetch data from the `IOlistener` .

10. The virtual CUDA library gets the response message from the forwarding module.

11. The `cudaGetDeviceCount(..)` function in the virtual CUDA library returns any output parameters and the return type exactly according to the "real" function.

The forwarding module is the last component of the system and is responsible for executing CUDA function calls natively using the real CUDA library. It is described in subsection 4.2.3.

Code example 4.11 on the following page shows the fake implementation of the `cudaGetDeviceCount(..)` CUDA API call.

Memory Transfers

The CUDA API provides many functions for handling memory transfers, the most basic being `cudaMemcpy(..)`. A range of other functions handle variations of 2 - and 3 - dimensional array data transfers as well. All of them involve source and destination pointers, as well as a property describing the direction of the copy; see code example 4.12 on the next page.

kind specifies the direction of the transfer, a total of four directions between the guest and device memory operating areas.

dst is the destination memory pointer. Whether it is a device or host pointer depends on the direction of the copy.

src is the source memory pointer. Whether it is a device or host pointer depends on the direction of the copy.

count specifies the number of bytes to transfer.

Our implementation of this function is fairly simple. Depending on the copy direction, the data is transferred from the guest or device piggybacked on top of the request or response message (see Figure 4.4 on page 70). The `size` parameter of `cudaMemcpy(..)` is important to determine the size of the

```
__host__ cudaError_t CUDARTAPI cudaGetDeviceCount(int *count){

    /* Prepare the request structure */
    cudaError_t respError = cudaErrorApiFailureBase;

    struct cudaGetDeviceCountStruct *msg_p =
        (struct cudaGetDeviceCountStruct*)
        malloc(sizeof(struct cudaGetDeviceCountStruct));

    msg_p->callheader.head.cmdType = normCall;
    msg_p->callheader.callID = facudaGetDeviceCount;

    msg_p->count = *count;

    /* Send the request to the host */
    if(
        sendMessage((void*) msg_p, sizeof(struct cudaGetDeviceCountStruct))
            ==
        FACUDA_ERROR)
        return respError;

    free((void*) msg_p);

    /* Get the response message and set any result parameters */
    if(recvMessage((void**) &msg_p) == FACUDA_ERROR)
        return respError;

    *count = msg_p->count;

    respError = msg_p->callheader.respError;

    free((void*) msg_p);

    /* Return basic error type */
    return respError;
}
```

Code Example 4.11: An example fake implementation of a CUDA API call.

```
cudaMemcpy(
    void* dst,
    void* src,
    size_t count,
    enum cudaMemcpyKind kind
    )
```

Code Example 4.12: The most basic API function for device memory transfers.

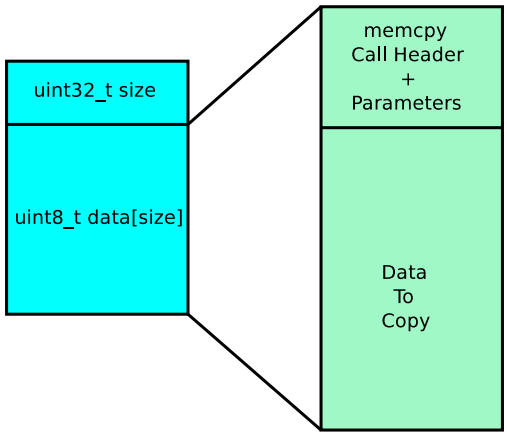

Figure 4.4: Wrapping of transfer data in message structures.

message, as the `void*` type doesn't denote the size of the data to be copied (it is just a pointer to a memory location).

Because of an issue discussed in Section 5.3.1, our implementation only implements the `cudaMemcpy(..)` call.

With the function forwarding implementation described, we can now continue with the management of kernels. This is the most complex part of the implementation.

Kernel Launches

We will now describe how kernels and kernel launches are managed by the virtual CUDA library.

Under the hood, kernel launches are fairly undocumented. The challenge in this area is to transfer the device code from the guest to the host and successfully execute it. To accomplish this it is important to understand how `nvcc` embeds device code and kernel launches into the executable. This section is fairly detailed because of the complexity involved in this task. The source code can be found in `lib/virtCudaLib.cpp`.

The implementation focuses on support for kernel launches using the ECS (refer to Section 2.3.1). The driver provided methods for kernel execution does not work from the guest, but changing this in the future should not be a big problem. Note that the forwarder must use the driver API to launch transferred device code.

The main problem of kernel launches is that CUDA and `nvcc` does a great job at hiding:

- The low level details of embedding the device code in the executable. We need to extract this code, send it to the forwarder and register it.
- The launching of kernels. We need to know when kernel launches happen so we can launch them at the correct time.

When a developer compiles a CUDA source file, `nvcc` takes care of compiling any device code to `PTX` or architecture specific code. This is embedded into the executable and not visible to the developer. Internally, it also keeps track of kernels so that they can be launched with the ECS.

The challenge here is the following:

- Find out how kernel launches are managed by `nvcc`.
- Find a way to handle compiled device code, so that it can be transferred to the host and executed.

Initial approaches was to use programs like `readelf`, `objdump` and `nm` to attempt to reverse engineer CUDA object files and executables. These programs make it possible to look at hidden functions, assembly code and data sections. It showed that under the scene, the ECS is translated to a specific set of functions: The CUDA runtime API functions for launching kernels. This means that the ECS is just another way to invoke another group of functions.

While this shows how kernel launches are handled, it provides no information on how the kernel code is embedded in the executable. Using the tools above to reverse engineer this is very time consuming, and in the end, `nvcc` itself gave us all the answers.

The simplest way to look at how `nvcc` compiles the ECS and manages kernel code is to use `nvcc`'s `--cuda` switch. This generates a `.cu.c` file that can be compiled and linked without any support from NVIDIA proprietary tools. It can be thought of as CUDA source files in open source C. Inspection of this file verified how the ECS is managed, and showed how kernel code was managed.

1. Device code is embedded as a fat binary object in the executable's `.rodata` section. It has variable length depending on the kernel code.

2. For each kernel, a host function with the same name as the kernel is added to the source code.

3. Before `main(..)` is called, a function called `cudaRegisterAll(..)` performs the following work:

   - Calls a registration function, `cudaRegisterFatBinary(..)`, with a `void` pointer to the fat binary data. This is where we can access the kernel code directly.
   - For each kernel in the source file, a device function registration function, `cudaRegisterFunction(..)`, is called. With the list of parameters is a pointer to the function mentioned in step 2.

4. As aforementioned, each ECS is replaced with the following function calls from the execution management category of the CUDA runtime API.

   - `cudaConfigureCall(..)` is called once to set up the launch configuration.
   - The function from the second step is called. This calls another function, in which, `cudaSetupArgument(..)` is called once for each kernel parameter. Then, `cudaLaunch(..)` launches the kernel with a pointer to the function from the second step.

5. An unregister function, `cudaUnregisterBinaryUtil(..)`, is called with a handle to the fatbin data on program exit.

It seems that `nvcc` keeps track of kernels and kernel launches by registering a list of function pointers (the second step in the list above). All the functions above are undocumented from NVIDIA's part. The virtual CUDA library solves the problem of registering kernel code and tracking kernel launches by reimplementing each one of them, which we will get to in a minute.

There is still one problem to solve, which is the size of the fat binary data object. This doesn't seem to be provided anywhere. You are able to see it in the source file, but not in the executable itself. We need this to be able to transfer the device code to the forwarder. A simple C program was written to look for elementary data types of variable sized, signed / unsigned

	Fatbin data size	bitfield [8:11]	Diff
One kernel	4688	4671	17
Two kernels	6312	6292	20
Three kernels	7936	7917	19

Table 4.1: The fat binary size field.

integers in a fat binary data object. This showed a data field in bytes [8:11] that closely, but not accurately, match the size of the object as an integer.

The only other alternative to finding the size of the fatbin data is to parse the data itself; however, the data structure of this file-type is not available. Thus the implementation makes the best possible use of the bit field to determine the size of the data.

It is now possible to extract kernel code and track kernel launches. Finally, the following list shows how the fake CUDA library manages kernel registration and launches. It does this by implementing these hidden functions (see `/lib/virtCudaLib.cpp`):

1. `cudaRegisterFatBinary(..)` gives as parameters a pointer to the fatbin data object. Our implementation of this function finds the approximate size by looking at the number in bytes [8:11], then sends this data to the Qemu forwarder.

2. `cudaRegisterFunction(..)` yields the kernel function pointer that is used to distinguish different kernel launches. This identifier is also sent to the Qemu forwarder.

3. When the guest attempts to launch kernels:

   - `cudaConfigureLaunch(..)` extracts the launch configuration and sends it to the Qemu forwarder. This includes block and thread dimensions et cetra.
   - For each `cudaSetupArgument(..)`, setup arguments are sent to the Qemu forwarder.
   - `cudaLaunch(..)` yields the identifier from the second step, which is sent to the forwarder. The forwarder retrieves all the launch data as described in 4.2.3 and attempts to launch the kernel.

Note that while configuring a kernel launch this way, the CUDA application may (and will) get interrupted and rescheduled at some stages. This is considered as a critical section in the program's execution, as other CUDA applications may also be attempting to launch kernels at that time. To address this, the forwarder associates launch configuration data with the calling guest Process Identifier (PID). The launch parameters are stored in linked lists, and can be retrieved when the guest makes to launch a kernel.

Automatic Code Generation

Our concluding remark on the implementation of the virtual CUDA library regard the amount of work required to implement everything. There are approximately 220 functions in the CUDA APIs, all of which require:

- An implementation in the virtual library for the guest.
- An implementation in the forwarding module in Qemu.
- Data structures for function call parameters and identifiers.

To address this problem we have developed a Python script that recursively parses the header files of the CUDA API for function declarations, starting with `cuda.h` and `cuda_runtime.h`. The script can be found in `python/parseFunc.py`.

When the script finds a function, the script generates the necessary implementation listed above in four separate files (`fake_functions.cpp`, `forwarding_functions.cpp`, `callID.h` and `structs.h`). These are placed in the `python/autogen` folder.

4.2.3 CUDA Forwarder

The final part of this section describes the forwarding module in Qemu. The task of this component is to forward function calls from VMs. The source code can be reviewed in `src/cudaforward.cu`. Refer to the `readme.pdf` file mentioned in Section 1.3 for a guide on adding `.cu` files to Qemu.

The forwarder registers a handler function with the `IOlistener` with the message handler shown in code example 4.5 on page 59. A simplified look at the implementation can be seen in code example 4.13 on the following page.

```
static void cudaFwdMsgHandler(
                          void *msg,
                          uint32_t msg_sz,
                          void** respMsg,
                          uint32_t *respMsgSz){

    if(msg == NULL) // Handle empty messages
        return;

    struct header *recHdr = (struct header*) msg;

    if(recHdr->cmdType == normCall){

        struct callHeader *recCallHdr = (struct callHeader*) msg;

#include "forwarding_functions.cpp"

    }
}
```

Code Example 4.13: Simplified message handler implementation.

Note the `#include` statement, where our autogenerated forwarding functions are added to the code.

The input / output technicalities are handled by the Python script described in Section 4.2.2. Code example 4.14 on page 77 shows a sample forwarding function.

We are now finished with our description of Vocale 1.x's implementation. The next section outlines the implementation of Vocale 2.x. The section is not as deep as this one has been, as the only big difference in the two versions is the way they implement data transfers.

4.3 Vocale 2.x

This section explains the new method for data transfers in Vocale 2.x between the guest and the host. The main components of Vocale 2.x can be seen in Figure 4.5 on the following page. Note the changes in the figure, which are to the data path components.

As we will see in Section 5.2.2, Vocale 1.x suffers from very poor bandwidth. In order to solve this, it was necessary to find more efficient ways to transport data between the host and the guest. Such transfers are important for all hypervisors to do for example clipboard operations and network packet forwarding. Qemu uses *Virtio*[25] to achieve this.

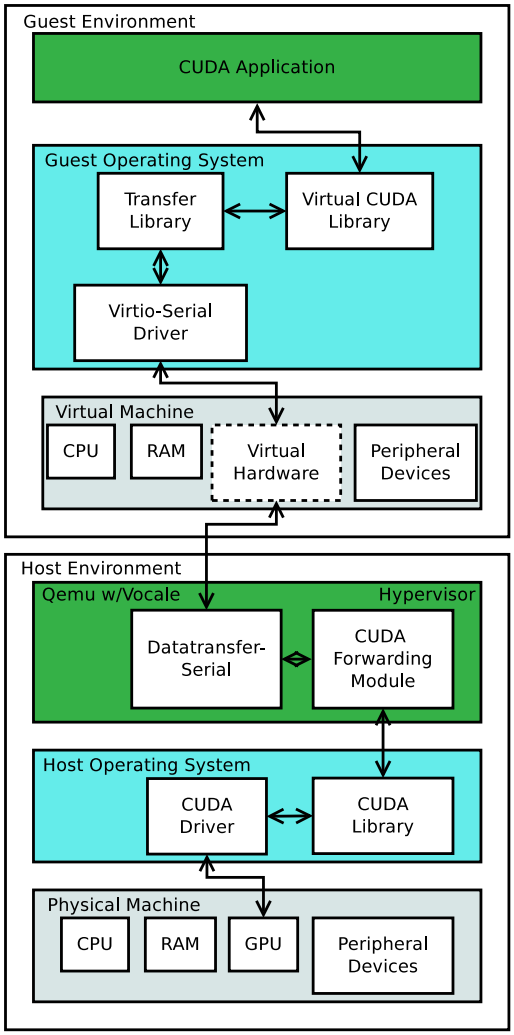

Figure 4.5: The implementation specific components of Vocale 2.x.

```
if(recCallHdr->callID == facudaGetDeviceProperties){

    /* Cast the message to the correct function parameter struct */

    struct cudaGetDevicePropertiesStruct *cudaGetDevicePropertiesStructVar =
        (struct cudaGetDevicePropertiesStruct*) msg;

    /* Make the call */
    cudaGetDevicePropertiesStructVar->callheader.respError =

        cudaGetDeviceProperties(
            &cudaGetDevicePropertiesStructVar->prop,
            cudaGetDevicePropertiesStructVar->device
            );

    /* Return the results */
    *respMsgSz = sizeof(struct cudaGetDevicePropertiesStruct);
    *respMsg = malloc(*respMsgSz);
    **(struct cudaGetDevicePropertiesStruct**) respMsg = *
        cudaGetDevicePropertiesStructVar;
}
```

Code Example 4.14: Function forwarding for the cudaGetDeviceProperties(..) function.

Virtio is a framework for virtual hardware adopted by hypervisors and OSs. The purpose of Virtio is to create a unified model for virtual hardware that can be shared between hypervisors and OSs. It also implements an efficient transport mechanism. Vocale 2.x attempts to achieve higher bandwidth for device memory transfers by integrating with a virtual hardware device, `virtio-serial`. `virtio-serial` uses the Virtio framework and shows more promising bandwidths. Refer to Figure B.2 and B.1 on page 131. These measurements show the bandwidth as a function of the transfer size between the guest and the host.

Note that the test is for illustrative purposes only - it was run using Linux pipes to access data on the host and guest side, which is a little different from what our Vocale 2.x does. It does show the potential of the underlying transport mechanism, however.

Vocale 2.x migrates to Qemu 1.0, because the `virtio-serial` interface of the previous version (0.15) had become out of date. Note that all file references in this section is to the `vocale-2.x` folder of the source code tree.

```
qemu-system-x86_64 -device virtio-serial -device datatransfer-serial,id=
    dtransf_test,name=datatransfer_serial
```

Code Example 4.15: An example invocation of the `datatransfer-serial` device.

```
static TypeInfo serialDeviceInfo = {

    .name           =   "datatransfer-serial",
    .parent         =   TYPE_VIRTIO_SERIAL_PORT,
    .instance_size  =   sizeof(DataTransfer),
    .class_init     =   dtransf_class_init,
};

static void dtransf_register_types(void){

    type_register_static(&serialDeviceInfo);
}
type_init(dtransf_register_types)
```

Code Example 4.16: `Datatransfer-serial` device registration.

4.3.1 Data Path Using Virtio-serial

The new data path implementation can be found in `src/hw/datatransfer-serial.c/.h`. It keeps the same interface as the `IOlistener` to ensure compatibility with the forwarder. The implementation uses Qemu's object model (described in Section 4.1) to create a new virtual hardware class, `datatransfer-serial`. Code example 4.15 shows how it is started from the command line.

`Datatransfer-serial` extends the `virtio-serial` class to be able to harness its bandwidth speed. The advantage of this is that the Virtio framework will handle everything on the guest side including a driver for the virtual serial device. The driver can be seen under `/dev/virtio-ports` on the guest. The implementation is inspired by another virtual hardware device, `src/hw/virtio-console.c/.h`, which does the same.

Code example 4.16 shows how `datatransfer-serial` registers itself as a new class type using Qemu's object model. It registers a class initialization function, `dtransf_class_init(..)`, that is called on command line initialization.

The initialization function is shown in code example 4.17 on the following page. Its main task is to implement functions inherited from `virtio-serial`.

```
static void dtransf_class_init(ObjectClass *klass, void* data){

    /* This example has been simplified; please refer to
     * the source code for the full example.          */

    k->init             =   dtransf_init;
    k->exit             =   dtransf_exit;
    k->guest_open       =   dtransf_open;
    k->guest_close      =   dtransf_close;
    k->guest_ready      =   dtransf_grdy;
    k->have_data        =   dtransf_gwrite;

}
```

Code Example 4.17: The `datatransfer-serial` device constructor.

These can be found in `virtio-serial`'s API, `src/hw/virtio-serial.h`. The following list describes this API.

dtransf_init / dtransf_exit are called when the VM is booted / shutdown.

dtransf_open / dtransf_close are called when the VM opens / closes the `virio-serial`'s device driver.

dtransf_grdy is called when the guest has is ready to receive data.

dtransf_gwrite is called when the guest writes data to the host.

When the guest writes data to the serial port, the `dtransf_gwrite(..)` handler is called with a pointer to the received data. Note that, for large transfers, it is not guaranteed that the data is sent in one chunk. Therefore, `datatransfer-serial` implements the same message passing flow as for Vocale 1.x. The difference is that data is no longer passed byte for byte, but in chunks of data with Virtio's transport mechanism. Any data transfer always starts with the size of the data to be received or transmitted. A buffer is immediately allocated in the host, to which data is transferred in chunks if necessary.

Code example 4.18 on the next page shows a simplified implementation of the received data handler. It will always assume that a message starts with the transfer length; after that, it will keep copying in data to the host until the transfer is done.

When the transfer is complete, a thread is created to execute the forwarder's message handler. The reason for doing this is simple - if we blindly

```
static ssize_t dtransf_gwrite(
                    const uint8_t *buf,
                    size_t len){

    /* Is this the start of a new message? */

    /* If so, initiate a new transfer. Allocate data
     * and set the size of the transfer. */
    if(transfState->transfSize == 0){

        uint32_t transferSize = *((uint32_t*) buf);

        transfState->transfSize = transferSize;
        transfState->data = malloc(transferSize);
        transfState->transfReceived = 0;

    }
    /* If not, continue reading in data. */
    else{

        for(i = 0; i < len; i++){
            transfState->data[transfState->transfReceived + i] = buf[i];
        }
        transfState->transfReceived += len;

        /* Are we done? If so, call message handler. */
        if(transfState->transfReceived == transfState->transfSize){

            struct replythread_args *replyargs =
                    malloc(sizeof(struct replythread_args));

            replyargs->port = port;
            replyargs->buf = (const uint8_t*)transfState->data;
            replyargs->len = transfState->transfSize;

            transfState->transfSize = 0;
            transfState->transfReceived = 0;

            pthread_create(
                &replythread,
                &rt_attrs,
                thread_reply,
                (void*) replyargs);
        }
    }
    return len;
}
```

Code Example 4.18: The `datatransfer-serial` received data routine.

called the message handler from this function, we can risk that the calling guest application never returns to read in the response. The message cycle thus never completes.

The thread implementation can be seen in code example 4.19 on the following page. The thread takes care of calling the forwarder's message handler, and sends the response message, if any, back to the guest.

The changes require some minor changes to the transfer library. These are reflected in `lib/transfer-virtio-serial.c` and are not covered here.

4.3.2 Changes to the CUDA Forwarder

During testing of Vocale 2.x it became evident that there was a new problem with the implementation. While function calls were forwarded and executed as normal, it seemed that writing data to device memory using functions like `cudaMemset(..)` had no verifiable effect.

The problem was related to CUDA contexts. We have described these in Section 2.3.1 as a form of software virtual memory solution: Device memory pointers allocated in one CUDA context have no meaning, and cannot be used, in another. What happened with Vocale 2.x was that, as reply threads returned, the context of that thread was lost. So was any device memory pointers allocated in any previous reply threads. This was an unforeseen issue that should have been part of the design.

A trivial patch was made to the forwarder to accommodate this issue (the changes are reflected in `src/cudaforward.cu`). The temporary solution is for the forwarder to have a single thread going at all times that handles any remote call execution. The message handler simply passes the call to the thread and waits for it to complete.

This ensures that contexts are not lost, but creates new problems: The solution is a serious security threat in that all guest CUDA applications now share the same context. We describe this issue in detail in Section 5.3.1.

In this chapter we have seen how Vocale is implemented. In the next, we will see how well the different iterations of Vocale work when compared to a real CUDA library. We will discuss our implementation and evaluate Vocale from a performance critical view.

```
static void* thread_reply(void* args){

    /* Call the data handler */
    datahandler(
            replyargs->buf,
            replyargs->len,
            &respMsg,
            &respMsgSz);

    /* Write the response message size */
    size = virtio_serial_write(
                replyargs->port,
                (const uint8_t*)&respMsgSz,
                4);

    /* Do we have response data? */
    if(respMsgSz != 0){

        /* Keep transferring until we are done */
        while(toSend != 0){

            while(virtio_serial_guest_ready(replyargs->port) == 0);
            justSent = virtio_serial_write(
                    replyargs->port,
                    ((uint8_t*)respMsg) + sent,
                    toSend);

            toSend -= justSent;
            sent += justSent;
        }
        free(respMsg);

    }

    free((void*)replyargs->buf);
    free(args);

    pthread_exit(NULL);
}
```

Code Example 4.19: The `datatransfer-serial` message callback function.

Chapter 5

Discussion and Results

This chapter evaluates the implementation and performance of Vocale. By doing this we want to find out if there are any particular challenges involved in implementing GPU acceleration for applications in VMs. We will also see what demands Vocale places on the hypervisor, and if the hypervisor can satisfy them. We will use the term *host* or *native performance* when talking about the performance of the real CUDA library.

The chapter layout is as follows. The first section explains how we have evaluated Vocale. We describe the programs we have run to verify that function calls, device memory transfers and kernel launches are working. We also argument for and describe the performance tests that we have subjected Vocale to.

The second section evaluates the performance of Vocale. We look at how well Vocale 1.x and 2.x works when compared to the performance of the real CUDA library.

The third and final section outlines our evaluation of Vocale. We look at how well our implementation of Vocale works in practice and describe the challenges and issues we have met during development and testing.

The performance tests can be found under the `performance_tests/` folder of the project source tree. All references to code in this chapter are to this directory.

5.1 Evaluating Vocale

To answer the goals of the thesis we need to find out how well the implementation of Vocale works and how efficient it is when compared to a real CUDA library. We also want to argue that our implementation is working correctly. In this section we describe how this has been done and the various tests we have implemented and run to assess the performance of Vocale.

We now explain how we can be certain that Vocale is indeed working.

5.1.1 Result Verification

In order to verify that the implementation works correctly, we have used NVIDIA's SDK examples to test CUDA API calls, kernel launches and device memory transfers. Note that we have not tested the entire API, as the objective of Vocale is not to create a perfect system, but rather study its performance impact and challenges. Stream related functions, for example, are not working (as we will discuss in Section 5.3.1). The following list describes the CUDA SDK example programs we have run on the guest to assess Vocale's correctness.

- **deviceQuery.** This is a small program that queries all GPUs installed on the system for various properties such as the number of CUDA cores, amount of global / shared memory et cetra.

  This has been verified working by comparing the output of the program from the guest with the output of the program from the native CUDA library on the host.

- **vectorAdd.** Sample CUDA program that shows how vector addition can be run on the GPU. It launches a kernel and performs device memory transfers between the host and device to do this. To verify the results, it uses the CPU to do the same vector addition, and cross checks the CPU results with those of the GPU.

  It has been verified working on the guest.

- **bandwidthTest.** This program can perform a range of functions to profile the bandwidth between the host and the guest. It uses the CUDA event API to measure the time it takes to perform memory transfers in all directions.

All the programs above have been tested and verified in both test beds (refer to Appendix A).

The virtual library also always returns the standard CUDA error code received from the forwarder, so the calling guest application will always know if something went wrong. The exception is if an error occurs that prevents the error code from being set (in other words, for some reason the call has not been executed by the forwarder). As an extra security, the error code is initialized with "unknown error" codes from the CUDA library to address this issue (see code example 4.11 on page 69):

- **CUDA_ERROR_UNKNOWN** is the CUDA driver API code for errors that are unknown to the CUDA framework.
- **cudaErrorApiFailureBase** distinguishes the same error as above, but for the CUDA runtime API.

One could of course create a new error code not implemented by CUDA instead of using these, but some support functions rely on them for correct operation. An example is the `cudaGetErrorString(error_code)`.

5.1.2 Performance Metrics

A virtual system should always provide at least the performance provided by its real counterpart. For Vocale, this is not possible, because all of Vocale's functionality is supported *by* its real counterpart. This means that for any function call, kernel launch or device memory transfer, the performance of that function is affected by the *extra overhead* incurred to execute the real counterpart of that function. Thus Vocale should provide *near native* performance instead, that is, minimize the execution overhead.

Therefore, our performance measurements should attempt to illustrate the extra overhead incurred for the range of functionality implemented by Vocale. We will compare Vocale's performance with the real CUDA library on the host, using the following performance metrics.

Inter / intra device bandwidth. We want to see the bandwidth to and from the device, as well as internally between memory regions in the device. There are examples of multi-core architectures that have failed simply because the bandwidth to, from or internally in the device is too slow.

Call execution delay. The time it takes from the execution of a particular API function until it returns.

Kernel Execution Delay. The time it takes from a kernel is executed to the time it is finished doing its work. This type of execution is slightly different from the call execution delay described above, because of the way Vocale manages fatbin data objects and launches kernels.

All the points above addresses various metrics that we will need to compare the performance of the virtual library with the real. The next subsection describes the performance tests we have developed to measure them.

5.1.3 Performance Tests

To be able to perform the measurements listed above, a number of Python scripts and C programs have been developed. Some of them are based on the `bandwidthTest` SDK example program described in Section 5.1.1. This subsection serves as documentation on how the tests are implemented and run.

A convenience script, `runTests.py`, has been made for the purpose of further work. It can be used to run the tests presented in this thesis on new hardware, or invoke them with different parameters. The script automatically analyses the results and creates plots. More information can be found by invoking the `--help` switch.

Device Bandwidth Measurement

This test can be found in `bandwidth_measurement/runTest.py`. Measuring device bandwidth is accomplished by the means of the bandwidthTest SDK example. The tests presented here perform device memory transfers in all directions, with transfer sizes up to 200 MB.

The bandwidthTest program usually works by invoking data transfers in a range; from start to end with a specific increment.

```
./bandwidthTest --device=0 --mode=range
                --start=1  --end=200000000 --increment=4096
```

When testing device transfers up to 200 MB, however, this approach can be impractical when the increment value is low. Because of this, the bandwidth measurement test invokes transfers on a *transfers per decade* basis.

For example, it can run 10 transfers per decade, meaning that in the transfer range 1 - 10, it will invoke transfer sizes {1, 2, ..., 9, 10}. This cuts the time required to perform the tests, and gives us a view of the whole transfer range at the same time.

Our measurements show the average bandwidth as a function of the transfer size. All tests have been run with 100 increments per decade, and 10 runs per transfer size.

Call Delay Measurement

The purpose of this test is to measure the time it takes from API calls are executed until they return. The measurement script is found in `call_overhead/runtest.py`. The test itself uses the CUDA API and is implemented as a normal CUDA program; see `call_overhead/call_overhead.cu`.

The CUDA test program uses a CPU timer to measure the time it takes between the execution of specific functions and kernels and the completion (return) of these operations. To ensure that the tests proceed without error, all tests are wrapped in C macros that perform error checking as well as time measurement.

The tests show the average execution delay on a set of CUDA API function calls. Each call has been executed 100 times for each test.

Virtual GPU versus Virtual CPU

The last of our performance metrics, the kernel execution delay, is measured with this test. During development and testing, however, we saw that it also gave a nice picture of Vocale as a whole. Therefore its usage is extended to also measure the efficiency of our virtual GPU contra the CPU.

Note that, in spite of the name, the purpose of the test is not to see which architecture can finish first. The purpose is to see how long time various steps in the Discrete Cosine Transform (DCT) algorithm takes, in a way that we can compare between a CPU and an GPU.

In this test, both the CPU and the GPU perform the DCT algorithm of randomly generated "image" frames of different sizes. The DCT is one of several stages involved in image compression. For example, an image frame can be split into 8x8 matrices of pixels. The DCT's purpose is to attempt to aggregate image information into the corner of the matrix so that its size can be reduced.

Following is the formula to calculate the DCT of an 8x8 pixel block. The example is taken from the `INF5063` course lecture on image compression[12][1].

$$G_{u,v} = \alpha(u)\alpha(v) \sum_{x=0}^{7} \sum_{y=0}^{7} (g_{x,y} - 128) \cos\left[\frac{\pi * u}{8}\left(x + \frac{1}{2}\right)\right] \cos\left[\frac{\pi * v}{8}\left(y + \frac{1}{2}\right)\right]$$

- $u \in [0,7]$, $v \in [0,7]$, $x \in [0,7]$, $y \in [0,7]$
- $G_{u,v}$ is the output value at location (u,v) in the current 8x8 matrix.
- $\alpha(i)$ is a normalization function.

$$\alpha(i) = \{ \begin{array}{ll} \sqrt{\frac{1}{8}} & i = 0 \\ \sqrt{\frac{2}{8}} & otherwise \end{array}$$

- $g_{x,y}$ is the input pixel value at position *(x,y)* in the current 8x8 matrix.

The calculation can be split into five stages. The test measures the individual time it takes to perform these, which we will now explain. For an input frame of m 8x8 blocks in the x direction, and n 8x8 blocks in the y direction:

Calculating a cosine lookup table. The cosine product can be viewed as a lookup table of 4096 entries, as it is determined by *x, y, u* and *v*, all of which range from 0 to 7.

Calculating a normalization table. The normalization product $\alpha(u)\alpha(v)$ can be viewed as a lookup table of 64 entries.

Matrix operations. For each 8x8 matrix in the input frame:

Normalization. Each pixel value in the current 8x8 matrix is normalized by subtracting 128 from it.

Summation and Multiplication. For each output pixel in the current 8x8 matrix, the two inner loop sums with multiplication in the cosine lookup table are performed.

[1]The subtraction in $(g_{x,y} - 128)$ is not a part of the DCT, but we include it to get more work for our test.

Output normalization. For each output pixel in the current 8x8 matrix, the loop sum from the previous step is multiplied with its corresponding value in the normalization table.

All the results of the DCT transformation are verified with a software library called FFTW[15]. The source code builds on the execution delay measurement test, and uses the same macros to perform error checking.

The test script can be found under `cpugpu_measurement/runtest.py`, and the source can be seen in the same folder under `cpuVSgpu.cu`. The measurements show the average execution delay of the steps involved in the DCT as a function of input matrix size. The tests have been run 100 times for each matrix size.

This was our concluding remark on how we evaluate Vocale. The next subsection will discuss the performance of the real CUDA library as it stands on the host, as well as Vocale 1.x and 2.x.

5.2 Performance Evaluation

In this section we study the performance of the real, non - virtualized CUDA library contra the performance of Vocale. The tests have been run in two different test beds, on three different GPUs (refer to Appendix A). Keep in mind that the `GTX 280` and `NVS 295` GPUs are installed on the same system.

5.2.1 Native Performance

This subsection presents the native performance results of the real CUDA library; that is, the CUDA library as delivered by NVIDIA running directly on the host. It is the same host that runs the guest. In this subsection we explain the behaviour of the graphs to the degree it is possible to get a picture of the natural properties of CUDA.

We will reference back to this subsection when we analyse the performance of Vocale.

Call Overhead

The call overhead measurements can be found in Figure 5.1 on the following page. Note that the y-scale is logarithmic in this example to get a better view of the measurements.

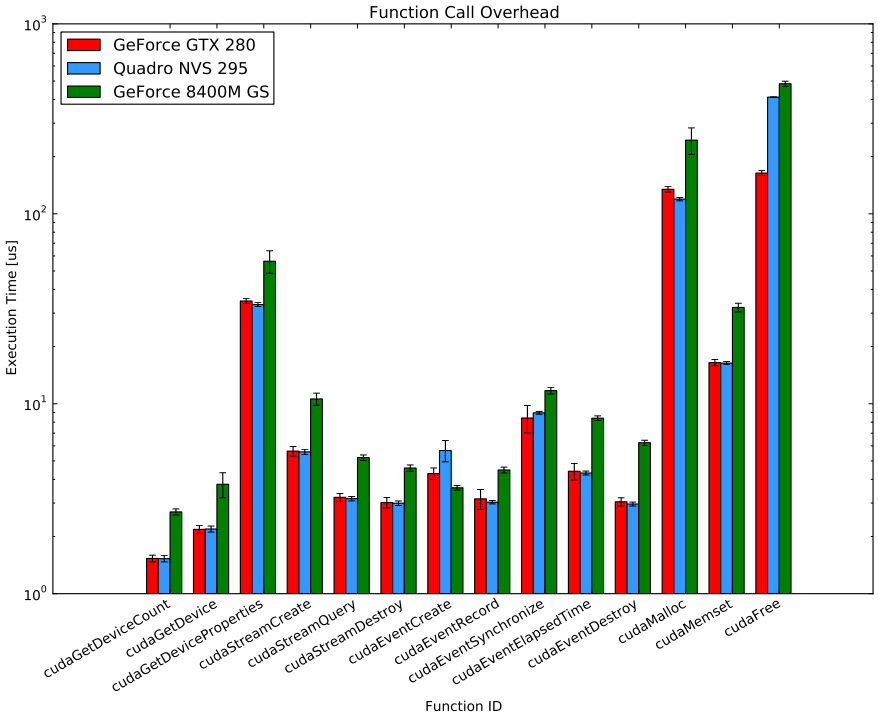

Figure 5.1: Native call overhead.

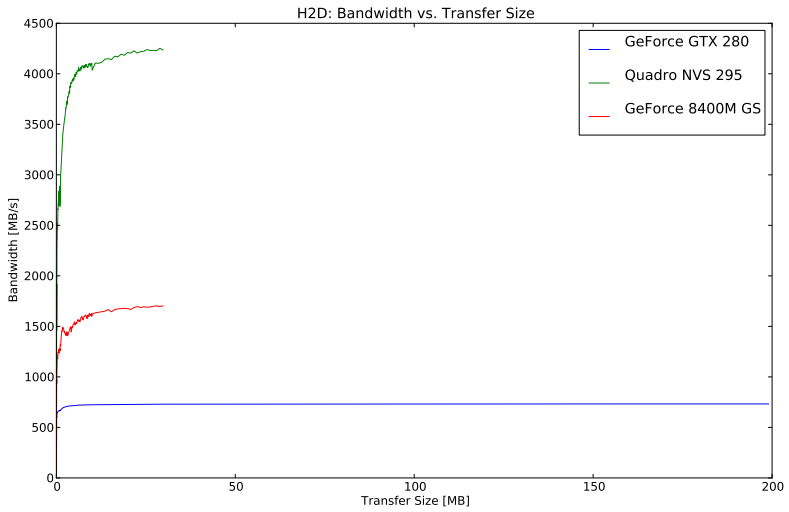

Figure 5.2: Native host to device bandwidth.

The measurements show the average execution delay of a selection of CUDA API calls. The standard deviation of the measurements can also be seen as the black line on top of the bars. We can see that some of the calls, namely `cudaGetDeviceProperties(..)`, `cudaMalloc(..)` and `cudaFree(..)`, have much higher overhead than the others.

We do not know exactly why this is. One can suggest that the function to get device properties uses a lot of time to query the selected device for attributes; as the output of the function is very extensive when compared to other calls. For the memory de/allocation functions, we do not know how CUDA copes with physical memory fragmentation. The high variation in execution time may be explained with this, but again, we do not know how CUDA handles these things internally.

Bandwidth Measurements

The bandwidth measurements can be seen in Figures 5.2, 5.3 and 5.4. These show the memory transfer bandwidth with respect to the transfer size.

For some of the plots, like the ones shown for the `GTX 280` GPU in Figure 5.3, the bandwidth can be viewed as sawtooth pulse. The interval between

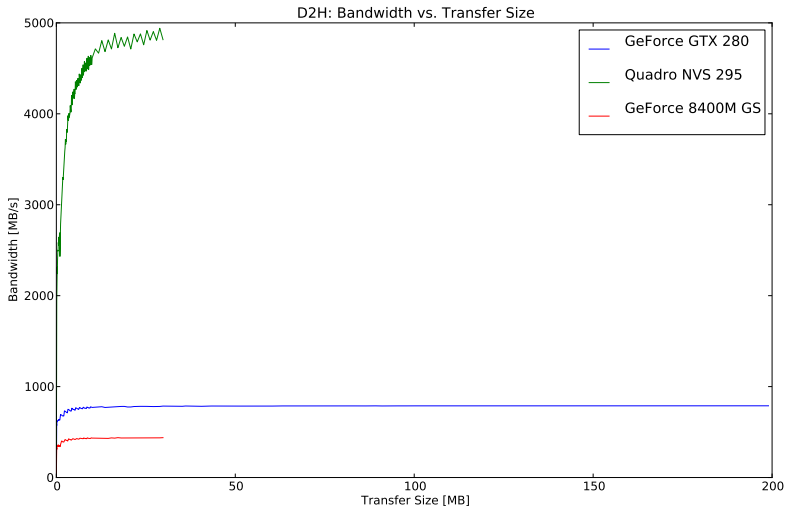

Figure 5.3: Native device to host bandwidth.

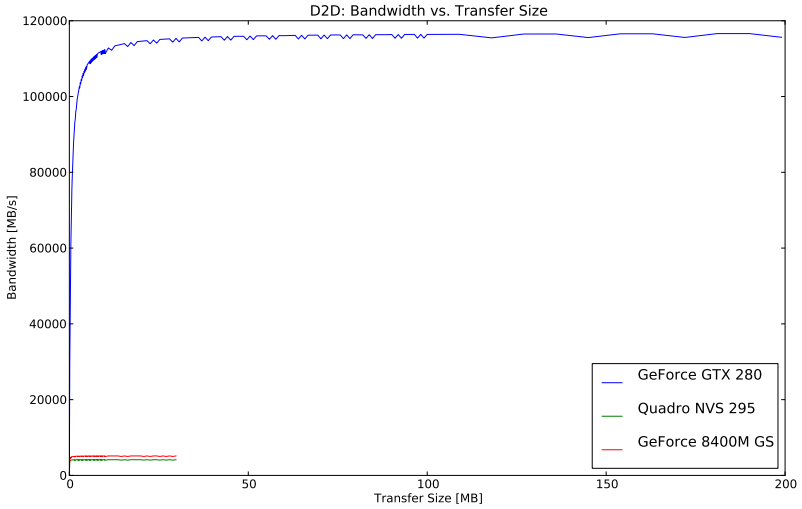

Figure 5.4: Native device to device bandwidth.

the teeth is more dense for lower transfer sizes. The reason for this property is that the test is run with some number x transfer sizes per decade (as described in the previous section) - in other words, the density of the transfer size measurements decreases as the transfer size increases.

Note that for two of the GPUs, the tests seem to stop at around 30 MB. This can be explained if we look at the measurements in Figure 5.4. For this particular transfer size, the test attempts to allocate $2 * 30MB$ of device memory to transfer a 30 MB memory block internally in the device. Thus the maximum transfer size of two of the GPUs, the `NVS295` and the `GF 8400M`, is around 60 MB.

This is strange, as both GPUs have, respectively, 128 and 255 MB memory (refer to Appendix A). The `GTX 280`, however, can run the tests with transfer sizes up to its full amount of memory, 1024 MB[1]. Our explanation for this is that GPUs that are rendering the display output, which is the case for the `NVS 295` and `GF 8400M`, reserve space in device memory that leaves only around 60 MB for CUDA operations.

Our final note on the bandwidth measurements are about the differences in bandwidth for the GPUs. For example, for the device to device measurements in Figure 5.4, it is observed that the `GTX 280` GPU excels when compared to the others. This huge difference can be explained by the number of CUDA cores (SMs) on each card.

The `GTX 280` has 240 cores, the `GF 8400M` has 16 and the `NVS 295` has 8. It makes sense that the differences in bandwidth can be explained by the number of cores, as the GPUs are tied for 1st, 2nd and 3rd place performance wise, respectively. This suggests that the CUDA library or driver somehow uses its SMs to perform copies, for example by executing memory copy kernels.

For the bandwidth measurements between host and device, there are also variations in bandwidth for the different GPUs. We consider this dependent on the system configuration for our two test beds.

CPU versus GPU measurements

The CPU versus GPU measurements can be seen in Figures 5.5 and 5.6.

Figure 5.5 shows the cumulative time taken to perform the DCT transformation, while Figure 5.6 shows a more detailed plot of the time taken to

[1]The tests have been run with a transfer size of up to 200 MB to be able to more easily see the measurements of the other GPUs.

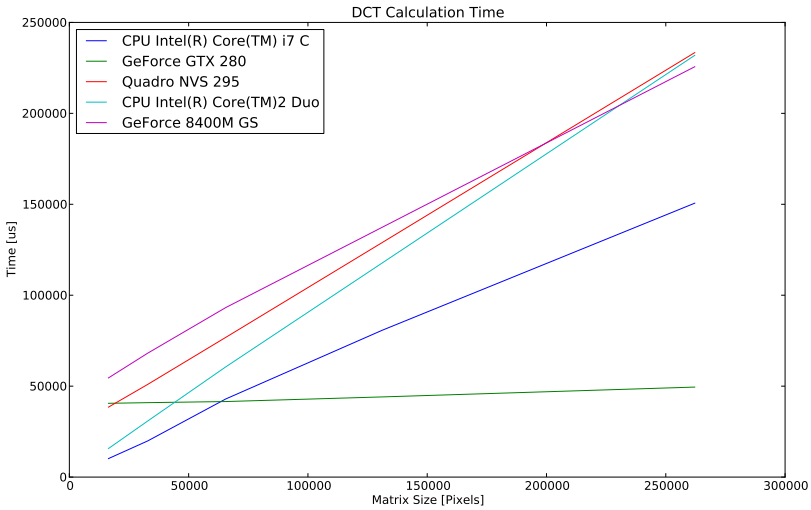

Figure 5.5: Native DCT calculation time.

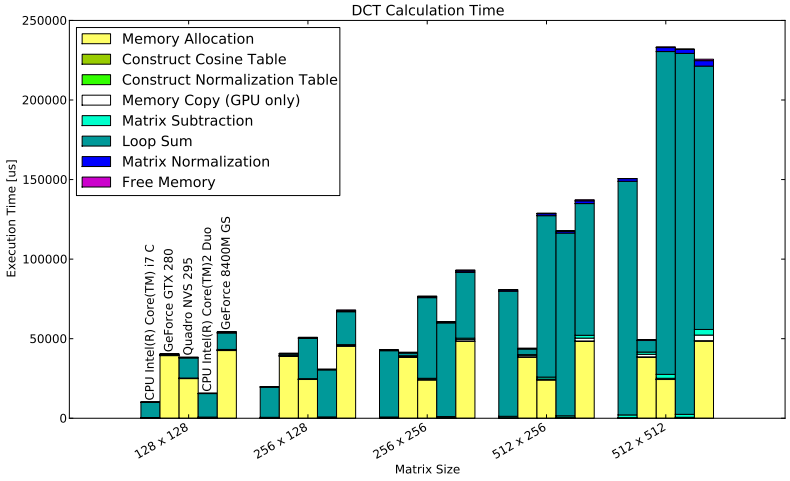

Figure 5.6: Native DCT calculation time (detailed).

perform various stages of the DCT. The time varies with the size of the input matrix. From top to bottom, the legend in Figure 5.6 shows the steps taken by the GPU and CPU, in sequence, to finish the DCT.

Figure 5.5 shows us that of all the GPUs, the `GTX 280` is the only one that beats both CPUs. Figure 5.6 shows that the loop sum stage of our DCT implementation dominates the runtime of the program - it is the step in the DCT algorithm that takes the longest time. The reason the `GTX 280` beats the CPUs and the other GPUs is that incorporates a large number of CUDA cores, so the loop sum stage is more efficient.

Notice the yellow bar of the plot, which shows the time taken to allocate memory. For all GPUs, this value is constant and very high when compared to the other steps. We raised this question on the NVIDIA forums [2], where we learned that the CUDA library performs initialization during the first CUDA API call, which is `cudaMalloc(..)`. We chose to leave the overhead here instead of ignoring it, as we think it is an important observation.

With the native performance results presented and explained, we can now move over to the performance of Vocale.

5.2.2 Vocale 1.x

This subsection discusses how well Vocale 1.x performs when compared to the host performance results described in the previous subsection.

Call Overhead

The call overhead measurements can be found in Figure 5.7 on the next page.

One of the first observations we can make of this test is the rather large overhead of `cudaGetDeviceProperties(..)`. This can be explained if we look at the size of the call header and parameters transferred with each call, shown in Table 5.1 on page 97. If we compare these sizes with our test results, we can see that the extra overhead of each call is directly connected to the size of its data structure.

From the results we can also see that the `GF 8400M` GPU has a much higher overhead than the other GPUs. This can be explained with the fact that the `GF 8400M` tests were run on a slower processor than the others (refer to Appendix A).

[2] http://forums.nvidia.com/index.php?showtopic=227656&st=0&p=1397764&fromsearch=1&#entry1397764

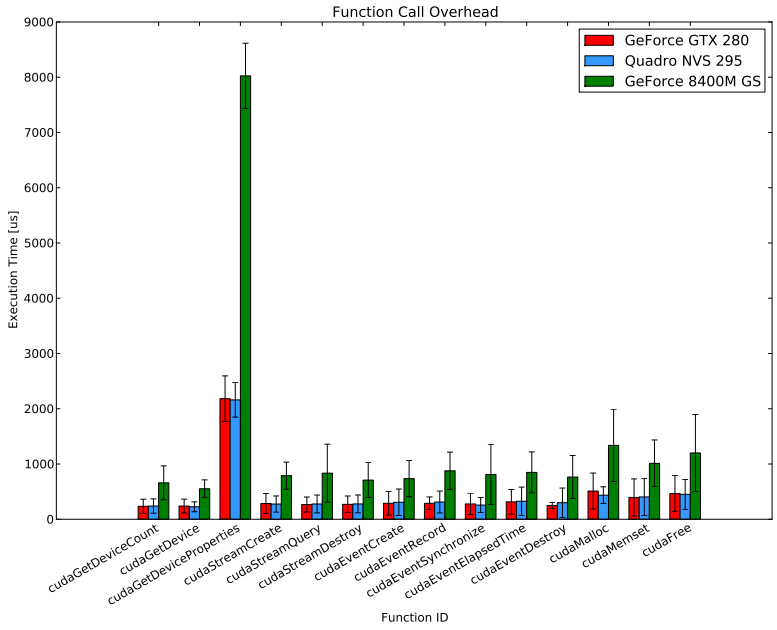

Figure 5.7: Vocale 1.x call overhead.

API Call Structure Size	
cudaGetDeviceCountStruct	24
cudaGetDeviceStruct	24
cudaGetDevicePropertiesStruct	504
cudaStreamCreateStruct	32
cudaStreamQueryStruct	32
cudaStreamDestroyStruct	32
cudaEventCreateStruct	32
cudaEventRecordStruct	40
cudaEventSynchronizeStruct	32
cudaEventElapsedTimeStruct	40
cudaEventDestroyStruct	32
cudaMallocStruct	40
cudaMemsetStruct	48
cudaFreeStruct	32

Table 5.1: Size of different call requests.

If we compare this result with the corresponding native measurements, we can see that the execution overhead is significantly larger in Vocale 1.x. It is in fact so large that the high native overhead of `cudaMalloc(..)` and `cudaFree(..)` is no longer visible. The overhead is incurred by the data path, so this already tells us that this implementation is very inefficient.

Memory Transfer Bandwidth

The bandwidth results for Vocale 1.x can be seen in figures 5.8, 5.9 and 5.10.

Comparing these with the native performance results makes it very evident that Vocale 1.x does not provide anywhere near the native bandwidth performance. In fact, it is so slow that the transfer sizes in the tests had to be limited to 1 MB. Both the guest to device and device to guest plots show bandwidths under 1 MB/s, while the corresponding native tests range from 500 to 4300 MB/s.

On the other hand, device to device bandwidth tests in Figure 5.10 shows more normal bandwidths for this transfer range. This gives us an important clue as to what makes transfers so slow. Let us consider the amount of copying involved when a guest user space program transfers data to the

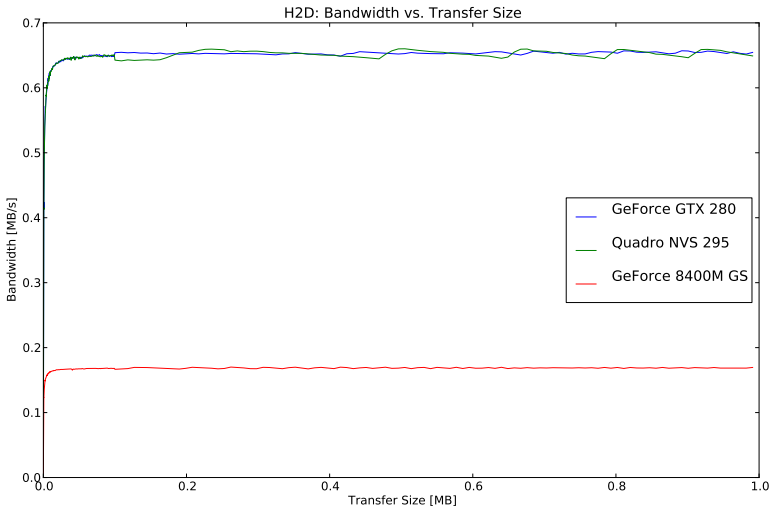

Figure 5.8: Vocale 1.x host to device bandwidth.

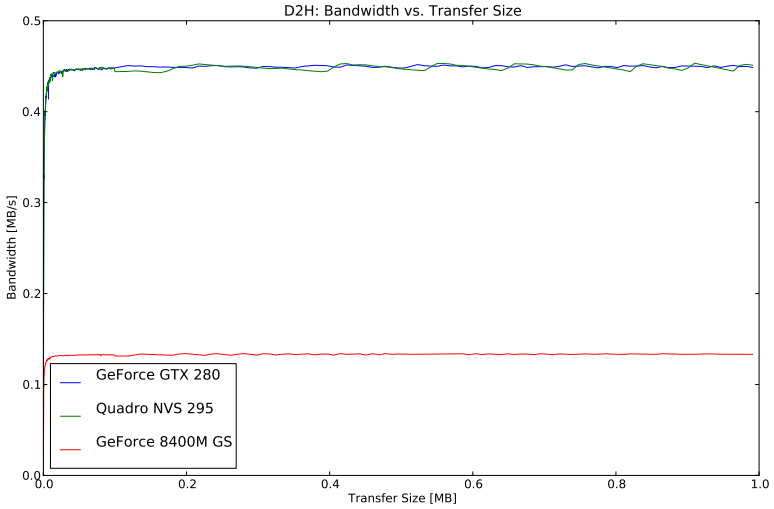

Figure 5.9: Vocale 1.x device to host bandwidth.

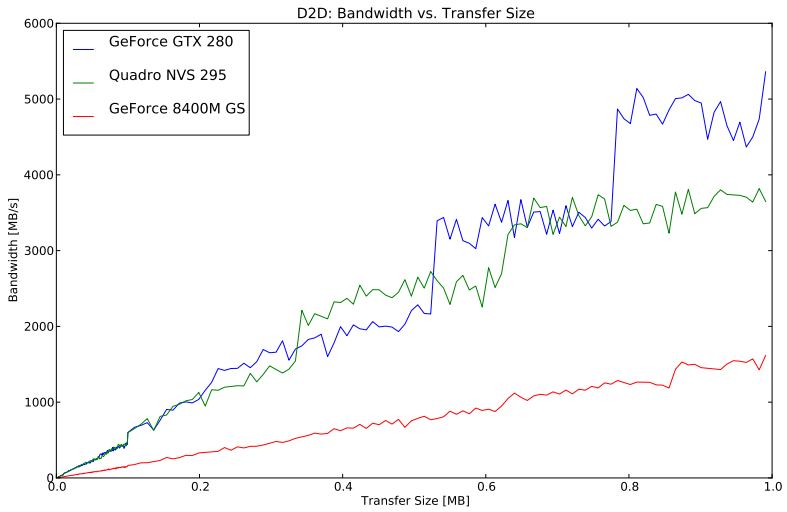

Figure 5.10: Vocale 1.x device to device bandwidth.

device:

1. The transfer request and data must be copied into the guest's kernel memory.

2. The guest copies the transfer request and data to the host through the `IOlistener` (see Figure 4.4 on page 70).

3. The forwarder executes the transfer with the data from the guest on the GPU.

That means three copies; the same goes for device to guest and device to device transfers. Now, what makes the device to device measurements so normal is the fact that they copy no transfer data between the guest and device - it is just the transfer request with some device memory pointers. Memory transfers between the device and guest, however, have to copy the transfer data with them as well, and the overhead incurred is very large. For a 10 MB data transfer, the `IOlistener` is called ten million times to transfer each byte. It is clear that the low bandwidth is caused by the port I/O involved in the second stage.

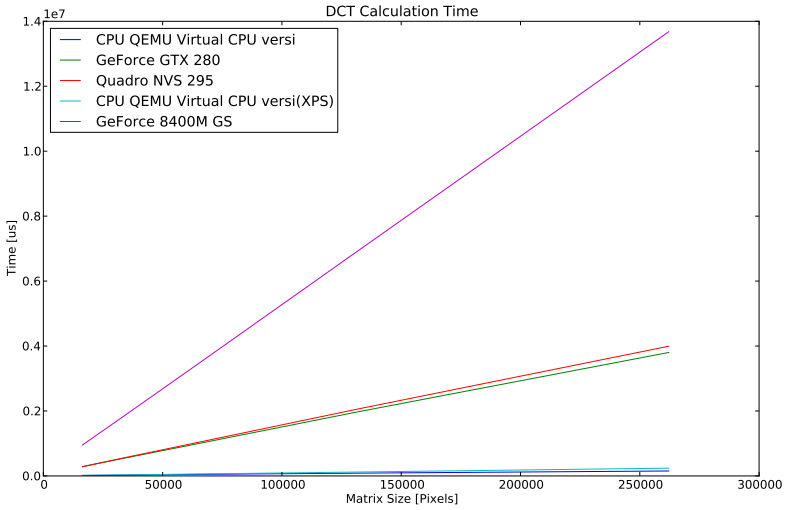

Figure 5.11: Vocale 1.x DCT calculation time.

CPU versus GPU

The CPU versus GPU measurements can be found in Figures 5.11 and 5.12. Note that the change in the two CPUs from the native measurements. This is because the CPU in these tests is emulated in a VM. The virtual CPU from the `GF 8400M`'s testbed is marked *(XPS)*.

Figure 5.11 shows us that Vocale 1.x fails to provide any GPU acceleration - the virtual CPU is more efficient in all cases. Figure 5.12 shows that memory copying between the device and guest dominates the execution time entirely. You cannot even see the kernel execution delay for which the test was intended.

We can see that it takes longer to perform device transfers with the `GeForce 8400M` GPU than the others. This matches the behaviour of the device transfer measurements in Figures 5.8 and 5.9.

On another note, these measurements also shows us that our implementation of the library is working, and that it is able to reliably execute several kernels in a guest CUDA application without breaking. It also shows that the implementation is correct in that the DCT results are verified as described in Section 5.1.3.

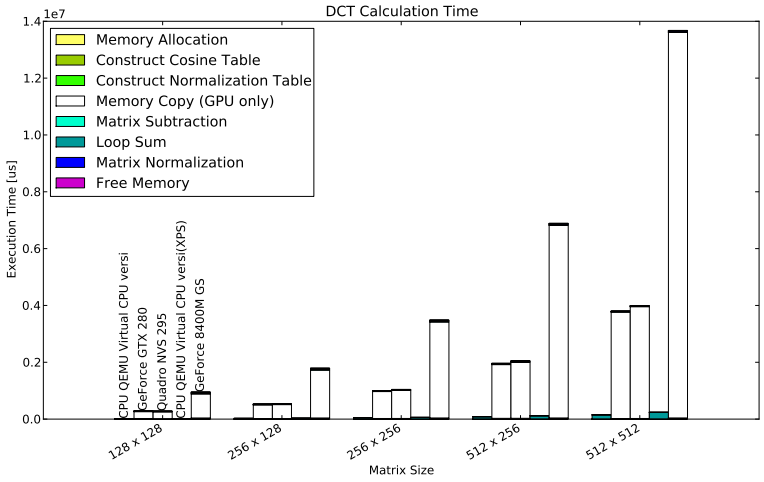

Figure 5.12: Vocale 1.x DCT calculation time (detailed).

With our study of Vocale 1.x's performance complete, we can now move to the performance of Vocale 2.x.

5.2.3 Vocale 2.x

The next subsections discusses the performance results of Vocale 2.x.

Call Overhead

The overhead measurements can be seen in Figure 5.13.

If we compare this test with the native one in Figure 5.7, we can see that the execution overhead is very similar to the one in Vocale 1.x. However, the time taken to execute the `cudaGetDeviceProperties(..)` call is significantly lower. This shows that we have accomplished faster data transfers.

Despite the fact that transfers are faster, there is still extra overhead incurred for CUDA calls. This overhead can be summarized in the following steps.

1. A calling CUDA application makes an API call.

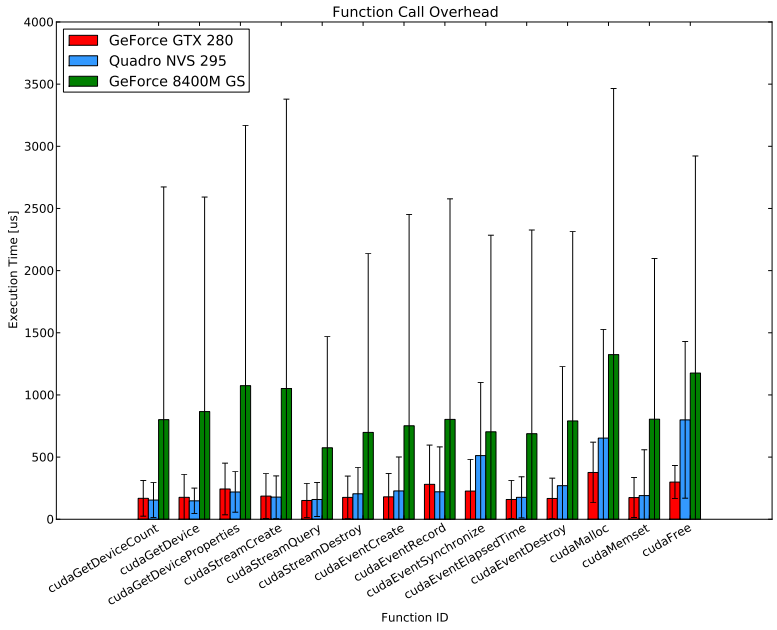

Figure 5.13: Vocale 2.x call overhead.

2. Allocating and preparing the API call data structure (refer to example 3.1 on page 48).

3. Sending the API call data structure to the forwarder.

4. Analysing the call header.

5. Executing the call with any given input / output parameters.

6. Sending the API call data structure back to the VM with return values .

7. Freeing the API call data structure and passing the results to the calling CUDA application.

The measurements give a picture of the time this process takes when compared to the equivalent native test. For example, if we consider the lowest native overhead calls in Figure 5.1 (those close to 0 μs), we can see that the overhead from the steps above is about 600 - 700 μs for the `GF 8400M`'s testbed and around 200 μs for the other.

Finally, our overhead measurements show us very high standard deviation values when compared to the native overhead tests. Inspection of our test data showed that the measured execution delay for the `GF 8400M` GPU sometimes jumps to 10000 μs, explaining the high values.

We don't know the reason for these jumps. It might be because of our new data path, which showed high standard deviation in Appendix B. We must, however, be careful with comparing Vocale 2.x to the test results in Appendix B. As we have mentioned before, these tests have been run by piping data in and out of the host, while Vocale 2.x implements data transfers differently. Regrettably, we did not have time to run a correct host to guest bandwidth test with Vocale 2.x.

Memory Transfer Bandwidth

The bandwidth test results of Vocale 2.x can be seen in Figures 5.14, 5.15 and 5.16.

We start by considering the guest to device bandwidth that can be seen in Figure 5.14. We can see that Vocale 2.x easily outperforms its predecessor, Vocale 1.x. It is still, however, far from the native performance seen in Figure 5.2. We can see that the `GTX 280` GPU is outperformed by the `NVS 295` GPU

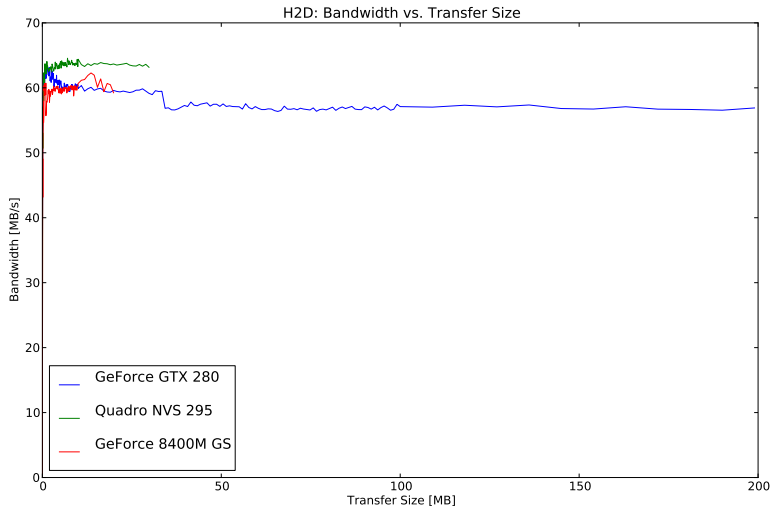

Figure 5.14: Vocale 2.x host to device bandwidth.

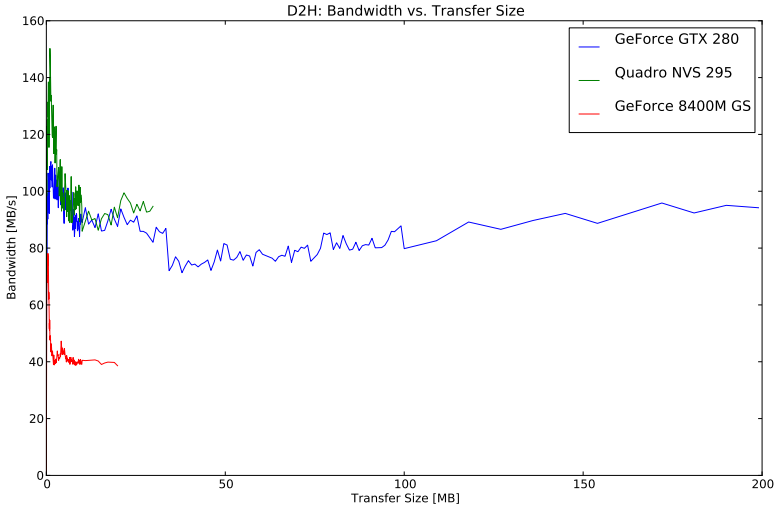

Figure 5.15: Vocale 2.x device to host bandwidth.

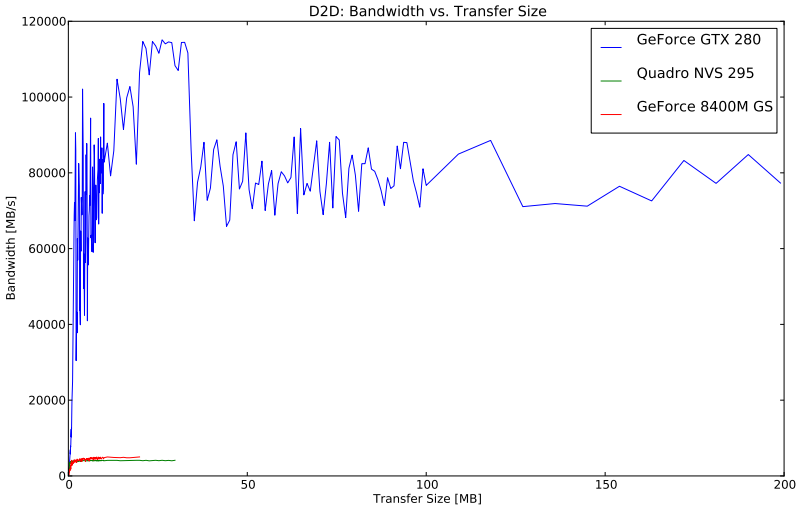

Figure 5.16: Vocale 2.x device to device bandwidth.

by a slight amount. This is because the `NVS 295` has a higher native host to device bandwidth (refer to Figure 5.2).

The device to host bandwidth of Figure 5.15 show the same trend. The plot is slightly more interesting, however, because the bandwidth is higher for low transfer sizes, and then degrades. We don't know why this is, but we believe that for small transfer sizes the bandwidth of `virtio-serial` is slightly higher.

Finally, Figure 5.16 shows the internal bandwidth of the GPU. As we discussed in Section 5.2.2, the results show a bandwidth that is much closer to native standards. When we compare this plot towards the native one in Figure 5.4, we can see that the variations in bandwidth is much more prominent for the `GTX 280` GPU in Vocale 2.x.

It is very peculiar that this GPU shows variations in the bandwidth that are so drastic, especially when comparing with the other GPUs. Again, we want to suggest that the reason lies in the number of CUDA cores on the GPU. Our hypothetical explanation is that the the internal data transfers of this GPU finish so fast that they are more susceptible to variations in transfer time from the standard deviation of Vocale's data path. This is, of course, again based on the assumption that Vocale 2.x's data path has a high

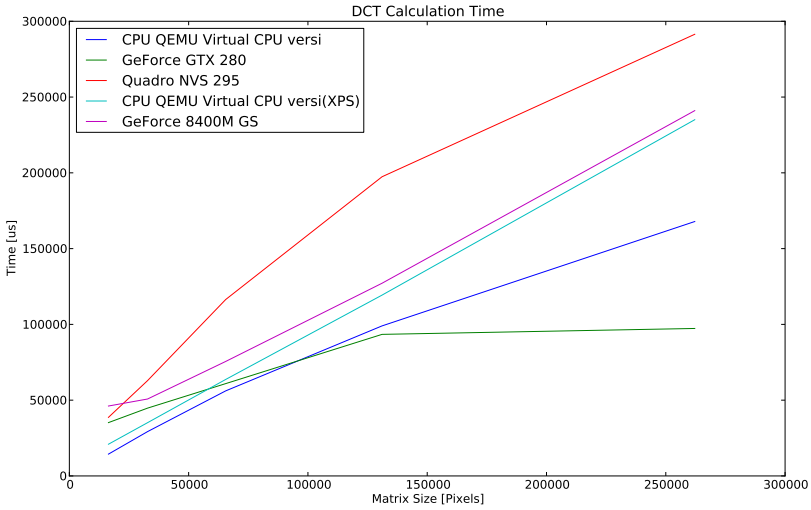

Figure 5.17: Vocale 2.x DCT calculation time.

standard deviation.

CPU versus GPU

Our final performance discussion is about the DCT calculation analysis. They can be seen in Figures 5.17 and 5.18.

Figure 5.17 shows us that Vocale 2.x is able to outperform the virtual CPU as the size of the matrix increases. This is true for the `GTX 280` GPU; the other GPUs simply do not have enough cores to process the inner loop fast enough.

We are now able to see the kernel execution delay as well. Compared with the native kernel execution delay, it is very close to host performance. This is logical as the kernel launches in Vocale are very low overhead - they are just a sequence of function calls. The actual transfer of the device code to the host is done at initialization, so it will not affect the launches itself.

Figure 5.18 shows the improvement of the device memory bandwidth. While it is still much higher than in the native measurements, we are now able to observe the other steps in the calculation as well.

Notice that the $5000\mu s$ memory allocation spike has apparently disap-

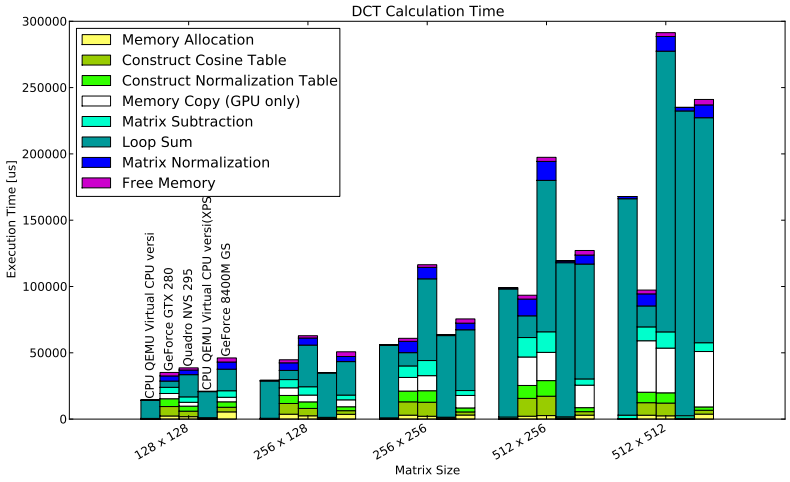

Figure 5.18: Vocale 2.x DCT calculation time (detailed).

peared from the measurements. In fact, it is still there, but as all of our measurements are run inside a CUDA process (the forwarder) that never stops between the test runs, there is just need for a single initialization. Thus, it will not show up in our measurements. Note that, even with the extra $5000\mu s$ execution overhead, the `GTX 280` will still be able to outperform the CPU.

5.3 Implementation Evaluation

In this section we discuss our implementation of Vocale and the observations we have made during development and testing. While Vocale gives VM applications access to the host's GPU, it is not perfect. Standard API calls, device memory transfers and kernel launches are working, but the functionality that is not supported also deserves some thought. By looking at the reasons things are not working, we will find out more about the difficulties involved in implementing systems like these and their requirements. It is also a a way of evaluating our design and implementation.

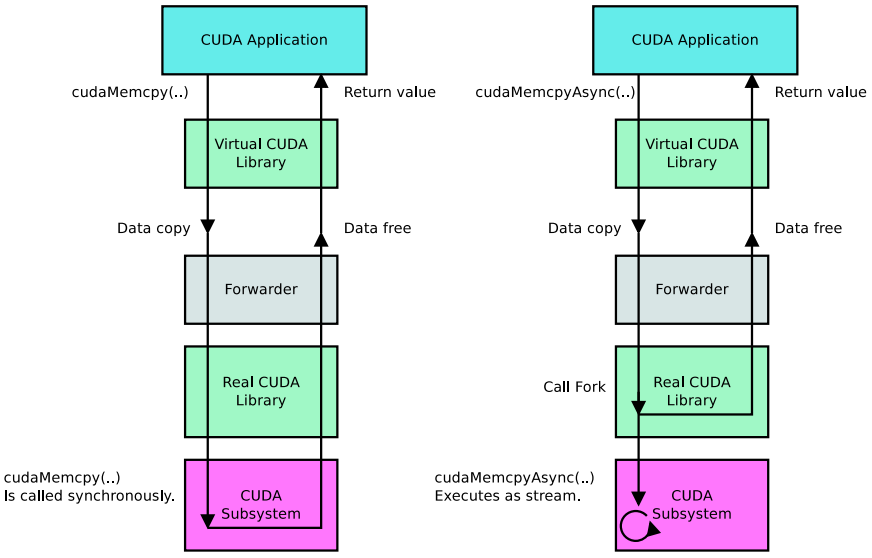

Figure 5.19: Executing a function synchronously or asynchronously using streams.

5.3.1 Vocale 1.x

In Vocale 1.x, our main concern was to get Vocale up and running. We wanted to implement the virtual CUDA library, find a way to transfer data between the guest and the host and last but not least, launch kernels remotely.

All these things represented a significant amount of work, and as a result we had not enough time to really make sure everything was working. In this subsection we will describe what we know is not working, and why.

Asynchronous Execution

CUDA provides streams to support asynchronous execution of functions (refer to Section 2.3.1). They are not supported by Vocale: Developers can use the stream management API to create, destroy and query streams, but they will not work in all scenarios. This is because of two main reasons.

The first reason is that data that is passed to and from the forwarding component in Qemu is not kept in memory after the return of any API call. This creates trouble for asynchronous calls that operate on user space data.

Consider the following example. The left part of Figure 5.19 shows how

a synchronous `cudaMemcpy(..)` call is handled. Once the call is intercepted by the virtual CUDA library, the call request and any data accompanying the call is copied into a new buffer that is presented to the forwarder (refer to Section 3.3 for a description of Vocale's components). The forwarder in turn executes the call on the real CUDA library, which handles the call and returns. Finally, the return data is sent back to the virtual library and the calling CUDA application, and the data buffer is freed in the process.

Now consider what happens when the CUDA application attempts to execute the asynchronous version of `cudaMemcpy(..)`; `cudaMemcpyAsync(..)`, in a stream. Refer to the right side of Figure 5.19. The call and accompanying data is transferred to the forwarder just like the synchronous call, but when the forwarder executes it, the call forks:

1. The calling thread immediately returns, passing back any return value to the virtual CUDA library and the calling CUDA application. The data buffer is freed.

2. Internally, the CUDA subsystem creates its own thread. This thread takes care of the execution of the asynchronous call.

It is easy to see what is going wrong here. Once the calling thread returns to the virtual CUDA library, the data buffer is freed. Asynchronous calls like `cudaMemcpyAsync(..)` keep working on this data - and creates a catastrophic memory access error that will crash the Qemu process.

The second reason streams are unsupported is that there is no *asynchronous event notification* mechanism implemented in the data transfer mechanism. In fact, the data transfer mechanism in Vocale 1.x only responds to calls from the VM. Qemu and the forwarder can do nothing to "wake up" the guest to transfer data.

Event notification is necessary for asynchronous functions which produce some kind of output to be copied back into the guest. Let us again consider the right part of Figure 5.19. The CUDA application running in the guest has just requested to copy some data from the device into a user space buffer using `cudaMemcpyAsync(..)`. The call progresses down to the forwarder in Qemu and forks. The calling thread returns immediately, while the CUDA library starts working on the data transfer in its own thread. Now, consider the situation when the stream is done and output has been produced. The data needs to be transferred back to the CUDA application, but because there is no way to notify it that data is available, it is lost.

Multiple Guest CUDA Applications

Vocale 1.x does not support multiple CUDA processes running in the guest. The reason is that Vocale was never designed to be used by several CUDA processes in the guest at the same time. However, this cannot be ignored in systems targeted for normal usage.

The reason multiple CUDA applications do not work in Vocale 1.x is a race condition. Let us take another brief look at how data is transferred between the virtual CUDA library and the forwarder.

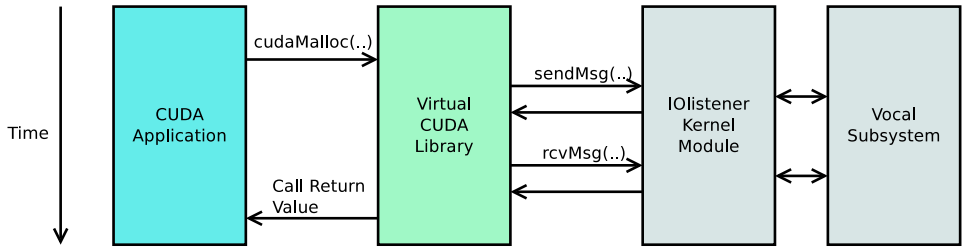

Figure 5.20: Call forwarding through Vocale's subsystems.

Figure 5.20 shows the steps taken by the virtual CUDA library to execute a function call remotely. When a CUDA call is intercepted, it sends a call identifier and any data accompanying the call to Vocale's subsystem by writing the `IOlistener` kernel module. Vocale's subsystem executes the call natively and prepares the result of the function call, which the virtual library retrieves by reading the `IOlistener` kernel module.

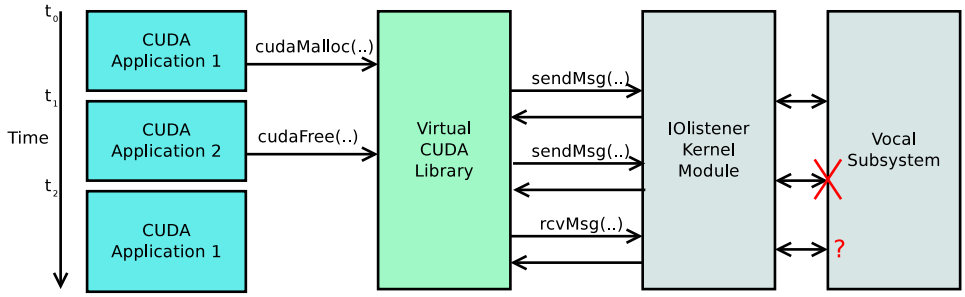

Figure 5.21: Race condition in Vocale.

```
semaphore_down(mutex)

sendMessage(call_request)
recvMessage(call_return)

semaphore_up(mutex)
```

Code Example 5.1: Example solution to Vocale's race condition.

Next, consider what happens if two CUDA applications attempt to execute functions at the same time. Figure 5.21 on the previous page shows such a scenario. At time t_0, a CUDA application calls `cudaMalloc(..)`, which is intercepted by the virtual CUDA library. The library just manages to send the request to Vocale's subsystem. At time t_1, a context switch occurs in the guest, and another CUDA process starts to run. This one calls `cudaFree(..)`. What happens next is that *another* function call request is landed in Vocale's subsystem. The result is that the subsystem doesn't know what to do - it is not designed to handle several requests in one go, but rather to serve a single threaded CUDA process in the guest.

This unfortunate problem can be easily solved by using a semaphore to ensure that whoever starts writing to the kernel module, also reads from it before anyone else can write it again. Code example 5.1 shows how this can be done.

Processes can now execute function calls randomly without having to worry about race conditions. However, a new problem regarding security becomes evident; that of contexts (refer to Section 2.3.1). Contexts exists on a per process basis, and provide some memory protection. For example, pointers to device memory will only have meaning inside the context it was allocated - processes cannot access device memory allocated from other contexts. Let us look again at how function calls are forwarded to illustrate the problem.

Figure 5.22 on the next page shows a system running several CUDA processes in both the host and the guest. The guest features two Qemu CUDA processes, 1 and 2, which use the virtual CUDA library. On the host, there are also two normal CUDA processes in addition to the Qemu process. The Qemu process can be viewed as a standalone CUDA process because it incorporates the forwarder (refer to Section 3.3), which executes all calls and kernel launches.

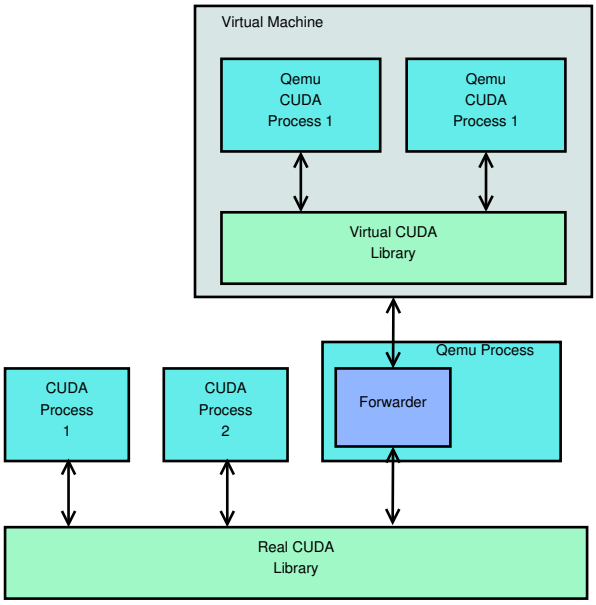

Figure 5.22: A system running several CUDA applications, both in the host and the guest.

The main problem here is that the two Qemu CUDA processes actually share a single context. In fact, any number of Qemu CUDA processes would share a context! The reason for this is that from the real CUDA library's perspective, the Qemu process is just a single process. When the forwarder executes calls on behalf of other processes, it uses a single context to execute all calls. This effectively creates a security hole, since all the guest CUDA processes can access any device memory allocated in the other processes. The problem is also valid for threads.

Fortunately the CUDA driver API contains methods to manage contexts, but fixing this problem also requires some fundamental changes to Vocale. For one, processes and threads must be tracked, so all forwarding requests must include the process and thread ID of the calling application. A system to keep track of currently active processes their threads must also be developed.

CPU Stall

Vocale 1.x makes relatively straightforward use of I/O ports to realise data transfers between the host and the guest (refer to Section 4.2). When the guest writes and reads data to and from certain I/O ports, handlers in the `IOlistener` track the transmission of data. When messages have been received, it calls the forwarder, which analyses the data received and executes functions and kernels as appropriate.

Unfortunately, this creates a problem with big data transfers which is best characterized as a form of CPU stall. Normally, when a CPU writes I/O ports, the value of that port is transferred immediately into one of the processor's hardware registers. In Qemu, however, a handler function is called in the `IOlistener`, so there is some extra overhead incurred when reading and writing I/O ports. No further investigation was made to find the root cause of this problem, but our hypothesis is that the I/O port handlers are run in the main CPU thread of Qemu, effectively halting all CPU operations in the VM until the handler is complete.

Automatic Code Generation

Section 4.2.2 discusses the implementation of the virtual CUDA library and addresses the problem that there are about 220 function calls that needs to be implemented as part of it. Each call requires an implementation in the host (forwarder), guest (as part of the library) as well as shared data structures (call structures and function identifiers).

A Python script was developed to address this problem. Our implementation takes a single header file as input, recursively searching any sub-included headers for function declarations. It successfully generates the following:

- An implementation of each function for the virtual CUDA library.
- Call identifiers and function parameter data structures.
- An implementation to execute each function from the forwarder.

Scripts like these should use common parsing techniques found in compilers to search for function declarations, but our implementation does not. It relies on a neatly written header where each new line is a new statement, and easily breaks with more messy header files. Fortunately, NVIDIA's developers did a good job here.

Now, aside from critiques, there were two key challenges that proved this job difficult.

Distinguishing input and output parameters. The return value of each library call is already used for error values, so any input / output parameters must be assigned in the parameter list. In C/C++, function parameters are copied onto the CPU's stack and lost when the function returns. Thus any output parameter must be denoted as a single or double pointer which points to the output variable in memory.

The void data type. Void denotes a data type which has no value. A common use is as a generic memory pointer (`void*`), for example for the device memory transfer functions in the CUDA API. The problem with this is that it can be both input and output parameter. Also, depending on the scenario, a size parameter may accompany the function. For this reason the memory transfer functions in the virtual library must be implemented manually. The alternative is to see if a pattern can be found to auto generate functions which involve parameters of the `void*` type.

This concludes our discussion of Vocale 1.x. The next subsection discusses the implementation of Vocale 2.x, which solves the two main issues of its predecessor.

5.3.2 Vocale 2.x

Our experiences with Vocale 1.x in the previous section show two main problems:

- The low bandwidth of memory transfers between device and guest. As we saw in our CPU versus GPU test of Section 5.2.2, these entirely dominate the performance of our DCT program.
- The CPU stall (refer to Section 5.3.1). This prevents users of VMs to actually be able to use their machine while performing device memory transfers.

The motivation of Vocale 2.x was thus to solve these issues. To address this, we moved away from our own implementation of port I/O data transfers

and used Qemu's own framework for virtual hardware and `virtio-serial` (refer to Section 4.3.1). The reason for doing this was our performance measurements show in Appendix B.

Our results show us, however, that while we get a good bandwidth increase from using `virtio-serial`, it is nothing close to the native bandwidth. It is also lower than expected, but this is because our Virtio measurements show the bandwidth between the guest and the host, and not the guest and device. There is thus an extra copy involved to copy data from the host to the device.

At any rate, using Qemu's Virtio framework for guest / host data transfers solved the CPU stall mentioned previously.

In this chapter, we have discussed the implementation and performance of Vocale. The purpose of this has been to find out how well Vocale was designed and implemented, and we also wanted to find out what kind of demands a system like Vocale places on its hypervisor. In the next chapter, we will make our concluding remarks on these matters.

Chapter 6

Conclusion and Further Work

In our final chapter we sum up our results discussion to make our final conclusive remarks. We answer our problem statement by looking at our experiences and results of Vocale.

The chapter layout is as follows. The first section describes the challenges related to the development of virtual GPUs, especially in the context of remote procedure calling of GPGPU libraries. These are closely intertwined with our implementation and performance evaluation in the previous chapter.

The second and final section outlines future work, where we propose solutions to the issues highlighted in the results chapter. We also present the start of Vocale 3.x, a shared memory implementation that was never completed. We sum up our work with an overall assessment of our work.

6.1 Challenges of GPU Virtualization

In our problem statement, we ask what the challenges of GPU virtualization are. This section attempts to answer that by using our implementation and performance results of Vocale.

6.1.1 Large Data Transfers

Our test results of Vocale 2.x show us that the device memory transfer bandwidth is very low compared to native standards. From our runtime analysis of the DCT calculation, we also see that GPU accelerated programs are easily outperformed by the CPU if they perform too many data transfers. This

renders the device memory bandwidth as an important bottleneck of Vocale 2.x.

In our work with Qemu we have not found a way to transfer data between the host and the guest that is efficient enough for GPU applications. The frameworks that exists are built for much smaller transfer sizes than those demanded by GPU applications - clipboard operations, packet transmission et cetra. We recognize the need for a new way to effectively share data directly between the guest and the host, that at the same time avoids reserving physical memory, as is GViM's approach.

Host / guest data transfers would be simple if not for the MMU of modern CPUs. While all the memory of a guest CUDA application essentially resides in the same physical memory as the hypervisor's, the MMUs in the host and the guest makes it impossible to access the memory directly between the two worlds. Memory is fragmented, swapped out to disk and changed at arbitrary times - in addition, a memory pointer in the guest has no meaning in the host, and vice versa.

This holds true for any design paradigm, front end or back end, of any virtual GPU used for general purpose computing. As long as the virtual GPU is backed by a physical one in the host, large data transfers will have to take place between the GPU and the guest. There is just no transfer mechanism in Qemu that supports this purpose.

6.1.2 Library Virtualization

One of the main reasons for choosing front end virtualization in the form of a virtual CUDA library was to avoid dealing in proprietary details. It was assumed that a virtual library is just a collection of function calls that can be easily forwarded through the VM layer. We predicted some work with resolving hidden functionality in the library. Our experience with Vocale proves this hypothesis wrong; in that not a little, but a great deal of time has been spent figuring out how to remotely launch kernels. Virtualizing a library like CUDA is therefore not as straightforward as one might think, depending on how well the internal functionality of the library is hidden. The issue of finding the size of the fat binary object in Section 4.2.2 is a prime example.

The need to automatically generate code quickly became evident during the implementation. As described in Section 5.3.1, the total number of functions in the CUDA API is around 220, each of which requires three separate

implementations. The main problems of this script is:

1. To distinguish input and output parameters.

2. To determine the size of the parameter data.

For example, the size of the data pointed to by `void*` pointers cannot be distinguished with operators such as `sizeof(..)`, and in the memory copy API functions it is impossible to determine whether the pointer is an input or output parameter. These functions must be handwritten.

The need for automatic code generation is also evident in that libraries such as CUDA are under constant development. At the time of writing, the CUDA tool kit has climbed to version 4.2, while the forwarder in Vocale uses 4.0. New releases deprecate and add functionality and this means work in maintaining the virtual library, asserting the need to automatically generate and remove the required code.

6.1.3 Process Multiplexing

In our evaluation of Vocale 1.x (section 5.3.1) we described a security problem in that all CUDA applications in the guest will share the same context.

This is an important observation which proves a point that should have been part of the design of Vocale. From the real CUDA library's point of view, the Qemu forwarder is *one CUDA process*, and all the CUDA applications running in the guest use that same process to perform their own work. This renders the forwarder as a *process multiplexer* in addition to being a pure call forwarder: The forwarder must execute API call requests from the guest as if it was running in its own process.

This is a relevant problem for any virtual library implemented with remote procedure calling to its real counterpart. As a result, the main call header data structure in Vocale should be extended with the *process ID* and *thread ID* of the calling guest application (see code example 6.1 on the following page. This makes it possible to track new processes and threads in the guest, and do the necessary work to multiplex each application in its own context.

A solution to this problem could be to mirror all the guest CUDA applications as child processes of Qemu in the host. Any function call or kernel launch executed by a given guest CUDA application could be forwarded to its "mirror process" in the host using pipes. This would make sure all guest applications have their seperate contexts.

```
struct callHeader{
    struct header   head;
    int             callType;
    int             callID;

    pid_t           PID;
    pid_t           TID;

    cudaError_t     respError;
    CUresult        drvRespErr;
};
```

Code Example 6.1: Vocale's call header extended with the process and thread ID of the calling guest application.

6.1.4 Architecture Independency

Our concluding remark on problems GPU virtualization must face is more classical. One of the benefits of VMs is the ability to emulate other processor architectures. For example, a user may want to run CUDA applications on a 32 - bit legacy processor, while the host processor is a standard 64 - bit processor. This can break systems that communicate data between the host and the guest. The use of standard types like `uint32_t` and `int8_t` from the `stdint.h` header is recommended, as we have demonstrated in some of Vocale's data structures.

The problem runs deeper than this for Vocale, however. We do not know the native size of data types used by the library; and these are exchanged between the host and the guest when we perform remote procedure calling.

There is a solution also to this problem. The CUDA library is wrapped in the *CUDA Toolkit*, which can be downloaded from NVIDIA's web pages. The tool kit comes in both 64 - and 32 - bit versions. This means that, to accommodate architecture independency, Vocale must offer both a 64 bit and 32 bit version of the virtual library - each of which are backed by their equivalent version on the host. Refer to Figure 6.1.

This is unfortunate as it will limit the otherwise flexible choice of guest processor architecture. Consider Table 6.1 on page 121. Notice that the third and seventh rows show no support for either library. The reason for this is that the equivalent CUDA library is not supported either in the guest or the host. Consider a guest emulating an Intel 64 bit processor with a host running a legacy 32 bit processor. The Intel 64 bit processor only supports

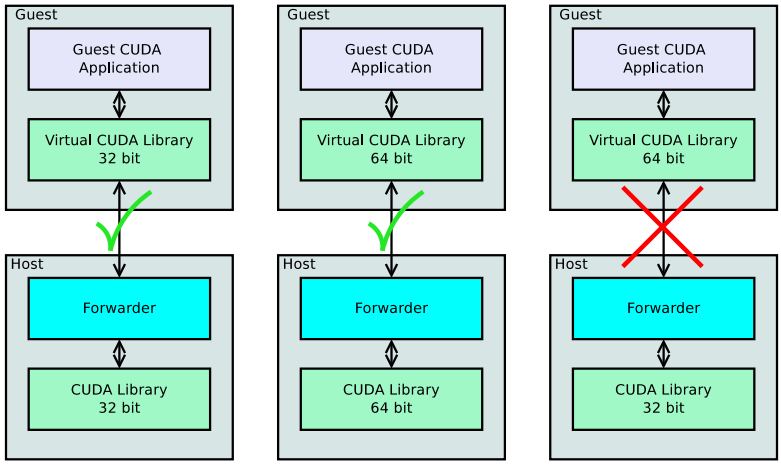

Figure 6.1: Valid and invalid combinations of host and guest CUDA libraries.

the 64 bit version of the virtual library, and the host only supports the 32 bit version. Hence, the two libraries cannot communicate unless they find a way to negotiate the size of their internal data types.

We have now made our concluding remarks to problems systems like Vocale must face. The last section of this thesis outlines further work, in particular a new way to perform device memory transfers.

6.2 Future Work

There are many possibilities for future work in the area of GPU virtualization. For Vocale, it is possible to extend the design and the implementation to provide a more complete system. For example, it is possible to look into multiplexing guest CUDA applications, find better ways to automatically generate code or find better ways to determine the size of the fat binary data objects in CUDA executables.

We consider the issue of device memory bandwidth as the most prominent, however. While we have proved that we can outperform the guest CPU with Vocale, guest CUDA applications will easily get outperformed if they do too many memory transfers.

If virtual GPUs are ever to hit the hypervisor world in full, there is a need to find a new, general purpose way to transfer large chunks of data efficiently

Host Architecture	Guest Architecture	Guest 32-bit CUDA library support	Guest 64-bit CUDA library support
x86	x86	Yes	No
x86	x86_64	Yes	No
x86	Intel 64	No	No
x86_64	x86	Yes	No
x86_64	x86_64	Yes	Yes
x86_64	Intel 64	No	Yes
Intel 64	x86	No	No
Intel 64	x86_64	No	Yes
Intel 64	Intel 64	No	Yes

Table 6.1: Architectural compatibility between Vocale's guest and host environments.

between the guest and the host. Ideally, we'd like to see such a mechanism as a part of the Virtio implementation. This way it could benefit virtual hardware in general, and not only our implementation of Vocale.

The subsection in this section outlines our start of such an implementation, inspired by Virtio[25] itself.

6.2.1 Zero Copy Data Transfers and Vocale 3.x

The Virtio article[25, Section 4.1] states the possibility for zero copy data transfers between hosts and guests as an alternative to its own data transfer mechanism. It is suggested that, for large data transfers, a technique called *page flipping* may improve the efficiency of the transfer. This has not been implemented in Virtio, but the article suggests for an enthusiast to prove the usefulness of page flipping. Possible future work for Vocale is to implement this and investigate if it can provide any acceleration over Vocale 2.x.

The intention of page flipping is to achieve zero-copy data transfers between the host and the guest. This is done by exploiting the fact that the physical memory of a guest can be accessed through the virtual address space of the hypervisor. Figure 6.2 on the following page shows such a scenario, where a guest CUDA application makes a request to copy some data to the GPU. The following list explains in short how we think this can be done.

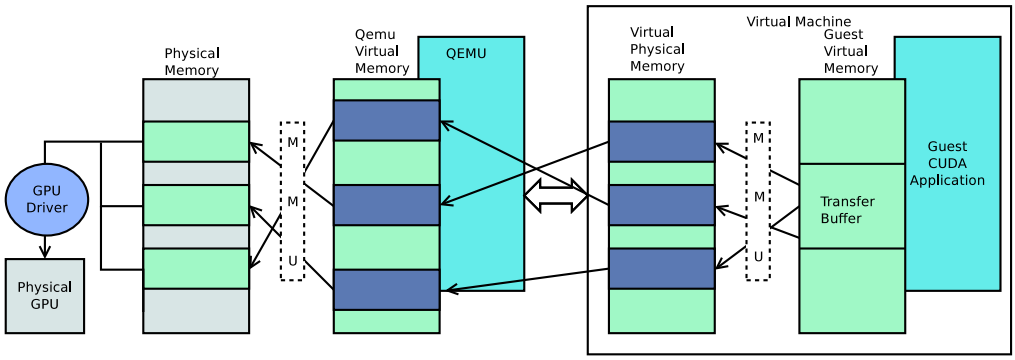

Figure 6.2: Zero copy data transfer from guest to host.

Our method is, of course, inspired by Virtio's remarks on page flipping.

1. A calling guest CUDA application initiates data transfer to a physical GPU on the host.

2. A driver in the guest receives a user space pointer to the data, and creates a special mapping called a scatter / gather mapping (refer to Section 2.5). This is a direct translation from user space virtual addresses to physical memory addresses. Depending on the physical memory fragmentation level in the guest, the length of the mapping increases or decreases. In our simple example, the CUDA application's transfer buffer is mapped to three chunks of physical memory (the guest's *virtual* physical memory). Scatter / gather mapped memory is pinned in memory: We cannot risk that the memory is swapped out to disk while we access it directly in the next steps.

3. The scatter / gather mappings are sent to a virtual hardware device that is part of the VMs hardware arsenal. This can be done through for example reserved physical memory areas implemented by PCI devices (Base Address Registers).

4. The virtual PCI device can now directly access the guest's physical address space by using a DMA API. This API lets us access the transfer data directly through Qemu's virtual address space.

5. Our virtual PCI device performs the copy to or from the GPU directly using streams to queue up the transfers.

6. When the streams and data transfer is complete, our virtual PCI device notifies the guest that the task is complete using hardware interrupts.

7. The driver in the guest releases the scatter / gather mappings. The CUDA application's data buffer is unpinned from memory, and the transfer is complete.

A folder named `vocale-3.x` can be found in Vocale's source tree. This is an unfinished implementation of page flipping that is inspired of a shared memory implementation called `ivshem` (refer to `src/hw/ivshmem.c`). We will not go into implementation specific details, as this is now out of scope for the thesis. We will, however, describe the basic concepts so that someone can pick up the work. The implementation consists of the following:

- A virtual PCI device called `zcpy`. Its source code can be found in `src/hw/zcpy.c/.h`.

- A new driver for the PCI device. The source code can be found in `driver/zcpy.c/.h`.

Basically, the following has been achieved:

- Automatically load the driver for the virtual `zcpy` PCI device on boot.

- Reserve physical memory areas that can be used for passing small amounts of data directly between the host and the guest; for example forwarding requests and scatter / gather mappings.

- The kernel module's user space interface is no longer implemented as read / write calls to the device file, but as `ioctl` commands[5, Chapter 6] (similar to system calls).

The idea of the `zcpy` device is to allow for mapping guest virtual memory addresses to the physical memory address space using scatter / gather mappings. These mappings can be sent directly to the `zcpy` hardware implementation in Qemu; from there, it is a matter of using Qemu's PCI DMA API to perform device memory transfers directly.

Anyone is free to continue the work, which should focus on implementing the following:

- Assigning hardware interrupts to the `zcpy` device to allow asynchronous event notification. There is an API in Qemu for doing this, and it is also possible to look at the implementation of `ivshmem`.
- Memory map `zcpy`'s reserved physical memory areas directly into the transfer library in the guest to allow zero-copy forwarding requests.
- Finish the implementation of scatter/gather mappings in the driver, to allow for mapping guest virtual memory to the guest's physical address space. The process of creating these mappings can be found in [5, Chapter 15].
- Verify and profile the bandwidth of the new implementation. It is not guaranteed that zero copy transfers will increase the efficiency of data transfers in Vocale. This depends solely on the implementation of the PCI DMA API and how fast it is, and is up to future work to find out. The PCI DMA API is necessary as the hypervisor has by default no access to the physical memory of the guest.

Our work with Vocale shows that VMs are able to benefit from GPU acceleration by emulating CUDA. We also see, however, that there are new challenges that needs to be solved for virtual GPUs to be a viable alternative to normal CPU processing in VMs. These challenges comprise issues of security, performance and maintainability. Once this is in place it is natural that VMs should provide GPU acceleration to their environment.

Our most important observation is that we can find no mechanism for data transfers in hypervisor environments optimized for large data transfers. GViM (refer to Section 2.6.2) solves this problem by reserving physical memory to allow zero-copy of data, but this approach can deplete system resources quickly. Nonetheless, we do not think there are any solutions that are faster: The next best thing is to perform page flipping, and that will demand some overhead in creating scatter / gather mappings, as well as pinning down memory.

We believe a combination of page flipping and the approaches to reserve memory taken by GViM will create a good solution, but this is for future work to find out.

Bibliography

[1] Virtio serial - features. Web.
http://fedoraproject.org/wiki/Features/VirtioSerial (26/4/2012).

[2] The gnu make manual. Web, July 2010.
http://www.gnu.org/software/make/manual/make.html (26/4/2012).

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37:164–177, October 2003.

[4] Wayne Carlson. A critical history of computer graphics and animation. Web, 2003.
https://design.osu.edu/carlson/history/lessons.html (26/4/2012).

[5] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux device drivers*. Number 3rd ed. O'Reilly, 2005.

[6] NVIDIA Corporation. Cuda api reference manual. Web.
http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_Toolkit_Reference_Manual.pdf (26/4/2012).

[7] NVIDIA Corporation. The cuda compiler driver documentation. Web. Included in Cuda Toolkit v.4.0 documentation.
http://developer.nvidia.com/cuda-toolkit-40rc2 (26/4/2012).

[8] NVIDIA Corporation. Cuda programming guide. Web.
http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf (26/4/2012).

[9] NVIDIA Corporation. What is cuda?
http://www.nvidia.com/object/cuda_home_new.html (26/4/2012).

[10] NVIDIA Corporation. Cuda best practices guide. Web, January 2012. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf (26/4/2012).

[11] Oracle Corporation. Oracle vm virtualbox: User manual. Technical report, Oracle, 2011. download.virtualbox.org/virtualbox/UserManual.pdf (26/4/2012).

[12] INF5063 Course. x86 and mjpeg presentation. Web, August 2011. http://www.uio.no/studier/emner/matnat/ifi/INF5063/h11/undervisningsmateriale/INF5063-03-x86-MJPEG.pptx (26/4/2012).

[13] R. J. Creasy. The origin of the vm/370 time-sharing system. *IBM Journal of Research and Development*, 25(5):483–490, 1981.

[14] Micah Dowty and Jeremy Sugerman. Gpu virtualization on vmware's hosted i/o architecture. In *Proceedings of the First conference on I/O virtualization*, WIOV'08, pages 7–7, Berkeley, CA, USA, 2008. USENIX Association.

[15] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[16] PCI Special Interest Group. I/o virtualization - address translation services. Web. http://www.pcisig.com/specifications/iov/ats (26/4/2012).

[17] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. Gvim: Gpu-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, HPCVirt '09, pages 17–24, New York, NY, USA, 2009. ACM.

[18] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One-level storage system. *Electronic Computers, IRE Transactions on*, EC-11(2):223 –235, april 1962.

[19] M. A. McCormack, T. T. Schansman, and K. K. Womack. 1401 compatibility feature on the ibm system/360 model 30. *Commun. ACM*, 8:773–776, December 1965.

[20] Susanta Nanda and Tzi cker Chiueh. A survey of virtualization technologies. Technical report, Stony Brook University, Department of Computer Science, 2005.

[21] Gerard O'Regan. *A Brief History of Computing*. Springer-Verlag London Limited, London, 2008.

[22] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. Gpu computing. *IEEE J PROC*, 96(5):879–899, 2008.

[23] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17:412–421, July 1974.

[24] Qemu. Getting started for developers. Web. http://wiki.qemu.org/Documentation/GettingStartedDevelopers (26/4/2012).

[25] R. Russel. virtio: towards a de-facto standard for virtual i/o devices. *SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[26] W. Sun and R. Ricci. Augmenting operating systems with the gpu.

[27] Andrew S. Tanenbaum. *Modern operating systems*. Pearson/Prentice Hall, Upper Saddle River, N.J., 2009.

[28] R. Uhlig, G. Neiger, D. Rodgers, A.L. Santoni, F.C.M. Martins, A.V. Anderson, S.M. Bennett, A. Kagi, F.H. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38(5):48 – 56, may 2005.

[29] David A. Wheeler. Program library tutorial. Web, 11. April 2003. http://tldp.org/HOWTO/Program-Library-HOWTO/ (26/4/2012).

[30] Xen Wiki. Xen store. Web. http://wiki.xensource.com/xenwiki/XenStore (26/4/2012).

[31] Wikipedia. Cloud computing. Web. http://en.wikipedia.org/wiki/Cloud_computing (26/4/2012).

[32] Wikipedia. Kernel-based virtual machine. Web. http://en.wikipedia.org/wiki/Kernel-based_Virtual_Machine (26/4/2012).

[33] Wikipedia. Qemu. Web.
http://en.wikipedia.org/wiki/QEMU (26/4/2012).

[34] Wikipedia. Virtualization. Web.
http://en.wikipedia.org/wiki/Virtualization (26/4/2012).

[35] Wikipedia. Vmware player. Web.
http://en.wikipedia.org/wiki/VMware_Player (26/4/2012).

Appendix A

Test Equipment

Vocale has been developed and tested on two architectures with three different GPUs.

		Dell XPS M1330	HP Compaq Elite 8100
Host Operating System		Ubuntu 10.10 x86_64	Ubuntu 10.10 x86_64
Guest Operating System		Ubuntu 10.10 x86_64	Ubuntu 10.10 x86_64
CPU	Type	Intel Core 2 Duo	Intel Core i7
	Intel-VT	Yes	Yes
RAM		3 GB	8 GB
GPU	Chip	GeForce 8400M GS[1]	[1]GeForce GTX 280 [2]Quadro NVS 295[1]
	GPU Memory [MB]	128	[1]1024 [2]255
	Memory Clock [MHz]	400	[1]1107 [2]695
	CUDA Cores	16	[1]240 [2]8
	Core Clock [MHz]	400	[1]1300 [2]1300

[1]Primary GPU (renders screen).

Appendix B

Virtio Performance

These measurements show the performance of `virtio-serial`, a virtual hardware device integrated in Qemu. The plots show the bandwidth as a function of the transfer size. The transparent area shows the standard deviation of the test results.

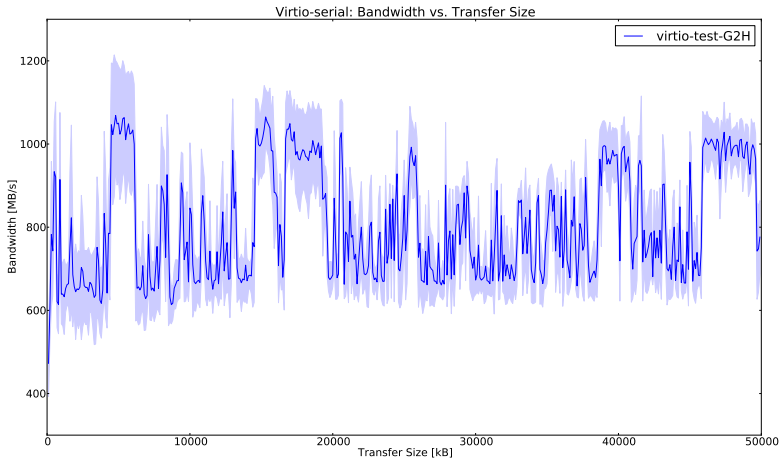

Figure B.1: Virtio bandwidth versus transfer size. Guest to Host.

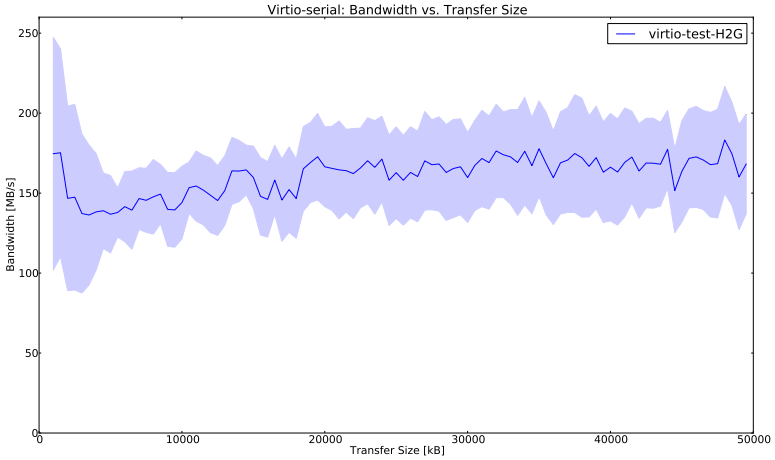

Figure B.2: Virtio bandwidth versus transfer size. Host to guest.